1

AD-A218 162

REAL-TIME CONCURRENCY CONTROL AND TRANSACTION SCHEDULING

By

RONNIE EDGE

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

1989

90 02 21 069

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS NONE
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S) AFIT/CI/CIA- 89-180
6a. NAME OF PERFORMING ORGANIZATION AFIT STUDENT AT UNIV OF FLORIDA; 6b. OFFICE SYMBOL *(If applicable)*	7a. NAME OF MONITORING ORGANIZATION AFIT/CIA
6c. ADDRESS *(City, State, and ZIP Code)*	7b. ADDRESS *(City, State, and ZIP Code)* Wright-Patterson AFB OH 45433-6583
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL *(If applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS *(City, State, and ZIP Code)*	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)* (UNCLASSIFIED)

REAL-TIME CONCURRENCY CONTROL AND TRANSACTION SCHEDULING

12. PERSONAL AUTHOR(S) RONNIE EDGE

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
THESIS/~~DISSERTATION~~	FROM ____ TO ____	1989	93

16. SUPPLEMENTARY NOTATION APPROVED FOR PUBLIC RELEASE IAW AFR 190-1
ERNEST A. HAYGOOD, 1st Lt, USAF
Executive Officer, Civilian Institution Programs

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL ERNEST A. HAYGOOD, 1st Lt, USAF	22b. TELEPHONE *(Include Area Code)* (513) 255-2259; 22c. OFFICE SYMBOL AFIT/CI

DD Form 1473, JUN 86 *Previous editions are obsolete.* SECURITY CLASSIFICATION OF THIS PAGE

AFIT/CI "OVERPRINT"

ACKNOWLEDGEMENTS

I wish to acknowledge the assistance and contributions of Professors Chow, Newman-Wolfe, and Chakravarthy. I thank Dr. Chow for his willingness to chair my committee and his continuous encouragement and support. I thank Dr. Newman-Wolfe for his comprehensive feedback, which has had a major influence on the organization of this thesis. I thank Dr. Chakravarthy for his inspiring comments and recommendations.

This thesis was supported by the United States Air Force through the Air Force Institute of Technology's civilian institution program.

TABLE OF CONTENTS

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

REAL-TIME CONCURRENCY CONTROL AND TRANSACTION SCHEDULING

By

Ronnie Edge

December 1989

Chairman: Yuan-Chieh Chow
Major Department: Computer and Information Sciences

Existing real-time concurrency control and scheduling techniques are inadequate for controlling the execution of transactions in a real-time database system. Some of these techniques place unnecessary restrictions on the concurrent execution of tasks in order to ensure they are executed correctly. Several of these mechanisms lack modularity--an important property in database concurrency control. Most of these real-time scheduling mechanisms lack the flexibility necessary to schedule both periodic and event-driven transactions to meet their timing constraints.

This thesis examines the desired properties of concurrency control and the desired properties and priority management issues of real-time transaction scheduling. A survey of some existing real-time concurrency control and scheduling mechanisms are presented and evaluated.

A new integrated real-time concurrency control and transaction scheduling mechanism, termed the dataflow scheduler, is presented. This scheduling technique meets all of the desired properties of concurrency control and provides more flexibility than existing real-time techniques. A roll-forward approach to data item conflict resolution is used by the scheduler. Simulation results show that this method performs better in meeting transactions deadlines than restarting or rolling back a transaction.

CHAPTER 1
INTRODUCTION

Real-time database systems are increasingly being considered for handling large quantities of data in systems for monitoring and control, tracking, and surveillance [Sha88b].

Existing real-time concurrency control and scheduling techniques are inadequate for use in real-time databases. These techniques place unnecessary restrictions on the concurrent execution of transactions. They also lack the flexibility needed to ensure the completion of a transaction, whether periodic or event-driven, by its deadline. In order to use real-time database systems for time-critical applications, integration of real-time scheduling and concurrency techniques is needed.

This thesis presents a new integrated concurrency control protocol and transaction scheduling technique for real-time database systems. This technique, termed the dataflow scheduler, provides more concurrency and flexibility than existing real-time concurrency control methods.

An important aspect of this scheduler is that it is dynamic. Transactions are scheduled to run when they enter the system. In this manner, the arrival times of transactions do not have to be known in advance. Transactions are scheduled to meet their deadlines based on their priorities. Event-driven transactions can be given higher priorities over periodic transactions. Because the dataflow scheduler schedules transactions when they enter the system, no restrictions are placed on modifying a transaction or changing certain characteristics of a transaction such as its period.

Another important property of this scheduler is that of predictability. The interaction of periodic and event-driven transactions can be simulated to determine scheduling feasibility.

This new approach is useful for scheduling transactions under timing and resource constraints in a centralized or distributed real-time database system.

Chapter 2 provides a discussion of the problems that can occur when transactions execute concurrently in a database system. The properties of concurrency control protocols needed to avoid these problems and other desired properties are presented. The most popular concurrency control techniques are also presented along with their drawbacks.

Chapter 3 discusses the desired properties of real-time transaction scheduling and the associated priority management issues of preemption, deadlock, and priority inversion.

Chapter 4 provides an overview of the existing real-time concurrency control and scheduling techniques available in the literature. Most of these techniques are being used in research-oriented distributed real-time systems. Each technique presented is also evaluated in the context of a real-time database system.

Chapter 5 introduces the dataflow scheduler--an integrated real-time concurrency control and transaction scheduling mechanism that uses a roll-forward approach to resolve data item conflicts. The properties of the dataflow scheduler are presented along with its priority management technique and failure recovery mechanism. Simulation results are presented and compared with that of restarting and rolling back a transaction to resolve data item conflicts.

Chapter 6 concludes by emphasizing the usefulness of the dataflow scheduler. In addition, areas in the dataflow scheduler and real-time database systems requiring further research are presented.

CHAPTER 2
CONCURRENCY CONTROL

Database concurrency control is the activity of coordinating the actions of transactions operating concurrently, accessing shared data, and potentially interfering with each other [Ber81, Ber87]. The goal of concurrency control is to prevent database updates performed by one transaction from interfering with database retrievals and updates performed by other transactions.

Problems

There are three problems that can occur from allowing transactions, executing concurrently, to have shared access to data [Elm89]. These three problems are the lost update, the temporary update, and the incorrect summary. The database concurrency controller must take measures to ensure that these problems are avoided.

The lost update. One of the more serious problems that can occur when transactions access shared data, concurrently, is the lost update [Elm89]. This problem occurs when two transactions update a database item in the database with one performing its update after access by the other but before the other transaction performs its update. To illustrate this problem, consider transactions T1 and T2 of Figure 2-1. Transaction T1 updates account1 by deducting 50 dollars from it. Transaction T2 performs another update by depositing 30 dollars into account1. If these two transactions are interleaved and executed as shown in Figure 2-2, where each line represents the temporal order of execution of transaction steps, the deduction of 50 dollars from account1 performed by transaction T1 is lost. Concurrency control protocols must produce schedules that avoid the lost update problem.

Transaction T1:	Transaction T2:
read(account1)	read(account1)
account1 := account1 - 50	account1 := account1 + 30
write(account1)	write(account1)

Figure 2-1 Example Transactions T1 and T2

Transaction T1:	Transaction T2:
read(account1)	
	read(account1)
account1 := account1 - 50	
write(account1)	
	account1 := account1 + 30
	write(account1)

Figure 2-2 Example Schedule of Transactions T1 and T2

<u>The temporary update</u>. Another problem the concurrency controller must avoid is the temporary update [Elm89]. This problem occurs when a transaction updates an item in the database and then fails prior to completing all of its updates. If other transactions are allowed to access the updated data item before it is changed back to its original

value, data inconsistencies can result. Consider transaction T3 of Figure 2-3. If transaction T3 failed after deducting 20 dollars from savings and before adding the 20 dollars to checking, then any other transaction accessing checking and savings will get incorrect values returned. Concurrency control protocols must not allow access to items updated by a failed transaction until failure recovery is complete.

```
Transaction T3:
------------------------
read(savings)
savings := savings - 20
write(savings)
read(checking)
checking := checking + 20
write(checking)
```

Figure 2-3 Example Transaction T3

<u>The incorrect summary</u>. Another of the more serious problems that may occur when transactions access data concurrently is the incorrect summary [Elm89]. This problem occurs when one transaction is calculating an aggregate summary function on a number of data items while another transaction is updating some of these items. Consider a transaction that calculates the difference between the total dollar amounts in the savings and checking accounts of a bank. If transaction T3 of Figure 2-3 were to execute after summation of all the savings accounts but prior to summation of the checking accounts, then the calculated difference between savings and checking will be off by 20 dollars.

Concurrency control protocols must avoid the incorrect summary problem.

Desirable Properties

There are three desirable properties of concurrency control protocols. These properties are consistency, correctness, and modularity [Sha88a]. Each property is determined from the possible schedules the protocol can produce. A concurrency control protocol is said to have the property of consistency if all possible schedules produced by that protocol do not violate database consistency constraints. The property of correctness is attributed to a concurrency control protocol if all possible schedules result in each individual transaction being executed correctly. Finally, the property of modularity is given to a concurrency control protocol if that protocol allows the correct scheduling of transactions without reference to the semantics of other transactions.

The properties of consistency and correctness are necessary to avoid the problems of the lost update and the incorrect summary; the temporary update problem is avoided by not allowing a transaction to make its updates visible to other transactions until it has committed [Elm89].

Consistency. Any valid concurrency control approach must ensure the database does not enter a state that has been defined by the designer to be inconsistent [Ber87, Esw76] Consistent states are defined by the boolean product of consistency predicates. A concurrency control protocol must not produce schedules that allow the database to enter a state where any of these predicates evaluate to false. An example predicate might be "the sum of funds in a savings and checking account must be preserved after execution of transactions transferring funds between them." Consider transactions T3 and T4 of Figure 2-4. Let the sum of savings and checking be the consistency constraint that must be preserved after execution of these transactions. If each transaction in Figure 2-4 were executed in a serial fashion (i.e., T3 before T4 or T4 before T3), the final state of the database would meet this consistency constraint. However, if we execute transactions T3 and T4 as shown in Figure 2-5, the database will enter an inconsistent state. In the schedule of Figure 2-5, the sum of savings and checking before execution of the transactions will not equal their sum after execution. Here the 20 dollars deducted from savings by transaction T3 is lost. In order to be effective, a concurrency control protocol must have the property of consistency.

Transaction T3:

```
read(savings)
savings := savings - 20
write(savings)
read(checking)
checking := checking + 20
write(checking)
```

Transaction T4:

```
read(savings)
deduct := savings * .05
savings := savings - deduct
write(savings)
read(checking)
checking = checking + deduct
write(checking)
```

Figure 2-4 Example Transactions T3 and T4

```
       Transaction T3:                 Transaction T4:
--------------------------------------------------------------
                               | read(savings)
                               | deduct := savings * .05
read(savings)                  |
savings := savings - 20        |
write(savings)                 |
read(checking)                 |
                               | savings := savings - deduct
                               | write(savings)
                               | read(checking)
                               | checking = checking + deduct
                               | write(checking)
checking = checking + 20       |
write(checking)                |
```

Figure 2-5 Example Schedule for Transactions T3 and T4

<u>Correctness</u>. A database concurrency control protocol must schedule transactions in such a manner that they are executed correctly. A concurrency control protocol is defined to be correct if all possible execution schedules generated by that protocol are correct. A schedule is considered correct if it is computationally equivalent to a serial schedule [Ber87, Cer84]. As an example, consider again the transactions depicted in Figure 2-4. Each transaction is now decomposed into tasks as indicated in Figure 2-6. Transaction T3 can be scheduled to execute with task A occurring before task B, task B occurring before task A, or both task A and B occurring pseudo-simultaneously (pseudo-simultaneity is used to characterize a pair of events when the temporal relationship between them cannot be determined [Cer84]). The result of each of these possible schedules is computationally equivalent to a serial execution of transaction T3. However, in the case of

transaction T4, the schedules where task D is executed prior to task C or pseudo-simultaneously may be incorrect. These two schedules cannot guarantee the amount added to checking will equal five percent of savings. It is critical for concurrency control protocols to have the property of correctness.

Transaction T3:	Transaction T4:

Task A:
read(savings)
savings := savings - 20
write(savings)
	Task C:
	read(savings)
	deduct := savings * .05
	savings := savings - deduct
	write(savings)

Task B:
read(checking)
checking := checking + 20
write(checking)
	Task D:
	read(checking)
	checking := checking + deduct
	write(checking)

Figure 2-6 Transactions T3 and T4 Divided into Tasks

Modularity. Modularity refers to the property of a concurrency control protocol that allows scheduling of transactions without reference to the semantics of other transactions [Sha88a]. A modular concurrency control approach is very important in database systems where new transactions can be added or existing transactions modified. If a database concurrency control protocol were not modular, adding a new transaction or modifying an existing transaction would jeopardize the correct execution of all transactions in that database system.

Serializable Methods

The most widely accepted basis for concurrency control in databases is that of serializability theory [Ber87, Sha88b]. If each transaction in a database system is correct and leaves the database in a consistent state when executed in some serial order, then any schedule that is equivalent to that serial schedule will also be correct and leave the database in a consistent state [Ber87]. Serializability is used as the definition of correctness for concurrency control protocols in database systems [Ber87, Cer84, Elm89, Kor86].

There are several concurrency control protocols used in database systems that are based on serializability. The three most popular protocols are two-phase locking, timestamp ordering, and optimistic concurrency control [Ber87, Cer84, Cou88, Elm89, Kor86].

Two-phase Locking

The most popular concurrency control approach in database systems is two-phase locking [Ber87]. Two-phase locking ensures the serializability of schedules by determining the order between every pair of conflicting transactions at execution time [Kor86]. The idea behind the two-phase locking protocol is simple. When a transaction accesses a data item in the database it is granted a lock on that item. Under two-phase locking, transactions do not release locks on data items until all locks are obtained.

This is where the name "two-phase" locking comes from. There is a growing phase in which locks are obtained and a shrinking phase in which locks are released. Shared access to data items is controlled through these locks. Transactions are granted a lock when granting the lock leads to a schedule that is serializable. Two-phase locking is a consistent, correct, and modular concurrency control protocol.

Even though two-phase locking is the most popular database concurrency control protocol, it does have some drawbacks. One drawback of two-phase locking is that it does not allow all schedules that are serializable and thus correct [Cer84]. To illustrate this drawback, look back at transactions T3 and T4 in Figure 2-6. Let's assume that savings and checking are at separate locations in a distributed database system. If the serial order were T3 then T4, task A would not release its lock on savings until task B is initiated and obtains its lock on checking. We see, however, that allowing task C to execute immediately after task A writes savings leads to a serializable, correct schedule.

Another drawback of two-phase locking is that it allows deadlocks to occur. Deadlocks occur when there is a circular wait for resources among two or more transactions [Pet86].

Still another drawback is that of cascading rollbacks [Kor86]. If a transaction aborts after releasing some of its locks, then all transactions that have accessed the released data items must be rolled back. This problem can be avoided by requiring transactions to hold on to their locks until committed. Even with these drawbacks, two-phase locking is still the most popular concurrency control protocol in database systems.

<u>Timestamp Ordering</u>

Another popular concurrency control protocol used in database systems is timestamp ordering. Unlike two-phase locking, timestamp ordering determines the serializability order in advance between every pair of transactions [Kor86]. Before a transaction is executed it is given a unique timestamp. If transaction T_i has been assigned a timestamp TS_i and a new transaction T_j enters the system, then TS_i must be strictly less than TS_j. The timestamps issued determine the serializability order. If TS_i is less than TS_j, then the scheduler must ensure that the schedule produced is equivalent to a serial schedule in which T_i appears before T_j.

In addition to the transaction timestamps, each data item has associated with it two timestamps called a write and read timestamp. The write timestamp is equal to the largest timestamp of any transaction that has successfully performed a write operation on that data item. The read timestamp is equal to the largest timestamp of any transaction that has successfully performed a read operation on that data item. Serializability of transaction execution is accomplished by ensuring read and write operations are done in timestamp order. Timestamp ordering is a consistent, correct, and modular concurrency control protocol.

As with two-phase locking, timestamp ordering has some drawbacks. Timestamp ordering cannot produce all possible serializable schedules [Kor86]. There are serializable schedules produced under two-phase locking but not timestamp ordering and serializable schedules produced by timestamp ordering but not two-phase locking [Kor86]. In timestamp ordering, serializability of concurrent transaction execution is achieved at the possible expense of restarting some transactions. If transactions are given access to updated data items before the updating transaction commits, cascading rollbacks can occur. Any time spent recovering from restarts is taken away from executable transactions.

Optimistic

In yet another protocol, termed optimistic concurrency control, the serializability order is determined at transaction completion. All updates to data items are accomplished on local copies kept by the updating transaction. There are three phases to the optimistic concurrency control approach. In the first phase, the transaction is executed. In this phase, the transaction is allowed to read data items with updates made on the transaction's personal copy. In the next phase, a check is made to determine if the transaction updates can be applied to the database without violating serializability. If the updates are serializable, then they are applied to the database in the final phase. If the validation is unsuccessful, then the updates are discarded and the transaction is restarted. Optimistic concurrency control is a consistent, correct, and modular concurrency control approach.

As with timestamp ordering, the optimistic approach has run-time overhead associated with transaction restarts. In systems where transactions are mostly independent, the extra level of concurrency achieved by this approach may compensate for the overhead of restarts. However, in large systems where transactions exhibit a large amount of dependency, the extra concurrency provided is outweighed by the cost of achieving it. Another problem with the optimistic approach is starvation [Cou88]. The optimistic approach alone does not guarantee that a transaction can complete. Although the likelihood of repeatedly restarting a transaction is small, some mechanism is needed to prevent its occurrence.

Nonserializable Methods

In some applications it may be acceptable to execute a schedule that preserves consistency even though it is not serializable [Gar83]. In distributed databases where data are located at many different sites connected by a communications network, performance considerations may dictate the use of a nonserializable concurrency control protocol.

Ensuring atomic transactions and serializable schedules in a distributed system has some drawbacks [Gar83]. First, transactions take longer to execute due to communication delays. Transactions that take longer to execute hold on to their resources longer, causing other transactions to wait longer to use those resources. Another drawback is the loss of site autonomy. Coordinating the commit exercise necessary to guarantee atomic transactions may cause sites to lose their autonomy. This occurs when a blocking commitment protocol is used. In a blocking commitment protocol, sites control access to their resources based on what is occurring at other sites. A failure at one site may unnecessarily tie up resources at another site. The objective of nonserializable schedules is to avoid the performance drawbacks described above and still meet the consistency requirements of the required application.

Garcia-Molina [Gar83] has introduced a nonserializable concurrency control method based on semantic consistency constraints rather than serializability constraints. The semantic consistency constraints are based on the observation that some database users may be satisfied when they access inconsistent data. Transactions that must have consistent data are called sensitive transactions. The author claims nonserializable, semantically consistent schedules provide more concurrency than serializable schedules.

Lynch [Lyn83] has defined a nonserializable concurrency control approach based on the relative atomicity of transactions. This relative atomicity (also termed multilevel atomicity) is defined by the activity of certain applications. In other words, in some applications it may be natural to expect transactions to receive inconsistent views of the data. Transaction atomicity is defined by breakpoints in the transaction where possible concurrent execution with other transactions can occur. When a transaction has no breakpoints, the execution of that transaction reduces to a serializable execution. In this way, a transaction can take on various levels of atomicity with respect to other transactions.

Sha et al. [Sha88a] have introduced a modular nonserializable concurrency control approach similar to that of Lynch [Lyn83]. Their approach decomposes both the database and the individual transactions accessing the database. The database is decomposed into atomic data sets using database consistency constraints. Each transaction is decomposed into elementary transactions. Elementary transactions preserve the consistency of the accessed atomic data sets when scheduled serializably with respect to the atomic data sets. In the situation where transactions and database cannot be decomposed, concurrency control reduces to that provided under serializability.

Even though nonserializable concurrency control methods may increase concurrency in distributed database systems, additional research is needed to determine their usefulness in distributed real-time database applications.

CHAPTER 3
REAL-TIME TRANSACTION SCHEDULING

Real-time transactions can be classified into two categories: time-driven and event-driven (also termed periodic and aperiodic). Time-driven transactions are executed periodically. An example might be a periodic update of certain equipment status information at each site in a monitoring and control database system for a communications network. An event-driven transaction is executed when a specific event occurs in the system. An example might be when a received signal level falls below a certain threshold at a microwave communications site in the network. The event-driven transaction could access specific site data to correlate a cause and recommend corrective action.

When dealing with a static system where only periodic, independent transactions execute and these transactions do not change, the feasibility of a schedule is simple and can be determined using the rate monotonic algorithm [Sha88b, Tok89]. A set of n transactions scheduled by the rate monotonic algorithm can always meet their deadlines if the following inequality holds:

$$C1/P1 + C2/P2 + . . . + Cn/Pn <= n(2^{1/n} - 1)$$

In the above equation, each Ci and Pi corresponds to the execution time and period of transaction Ti respectively. Given a periodic, independent transaction set, the above algorithm will determine if a feasible schedule exists.

The rate monotonic approach to determining a feasible schedule for transactions in a real-time database system is inadequate for a number of reasons. It lacks a mechanism for controlling the concurrent execution of transactions accessing shared data. Also, an effective real-time scheduler must be able to handle both periodic and event-driven transactions. The simple assumption that all transactions are independent (i.e., they do not depend on other transactions to accomplish their jobs and do not access shared data) is usually not valid. One should expect a real-time database to be a flexible system where transactions can work together and access shared data to accomplish their jobs. Also, one might expect real-time database transactions to be modified and new transactions to enter the system. The static scheduling algorithm of rate monotonic is inadequate because it cannot deal with event-driven transactions and transactions accessing shared data.

Properties

In addition to the concurrency control properties of consistency, correctness, and modularity, a real-time transaction scheduling mechanism must be capable of satisfying the timing constraints of transactions accessing the database. Consider our previous example of a monitoring and control database system for a communications network that required a transaction to update the status of critical equipment periodically. If such data were not kept up to date, the monitoring exercise would be a failure.

Flexibility. In order to be able to meet the timing constraints of both periodic and event-driven transactions, the scheduler must be flexible. A real-time transaction scheduler is considered flexible if it can schedule the highest priority transaction to meet its deadline regardless of the arrival time of that transaction.

Predictability. The behavior of a real-time transaction scheduler for a given transaction set should be predictable. This scheduling property of predictability is necessary to provide some confidence during the design stage that the system can meet the timing constraints of the transactions.

Priority Management Issues

When designing a real-time transaction scheduler, the designer must deal with the priority management issues of preemption, deadlock, and priority inversion. An effective real-time concurrency control protocol must manage transaction execution so that, whenever possible, transaction deadlines are met. Before discussing these issues in detail, a framework on data item conflict resolution is needed.

In some instances, it may be necessary for the scheduler to resolve a data item conflict between a higher priority task and a lower priority task. The conflicting lower priority transaction can be restarted, rolled back, or allowed to execute (rolled forward). Restarting a transaction requires undoing all of the work accomplished by the restarted transaction. Rolling back a transaction requires undoing that work accomplished after the transaction accessed the conflicting data item. Roll forward requires preemption of the higher priority task to allow the lower priority task to execute.

Preemption. The first major issue in scheduling real-time transactions is preemption. In systems where both event-driven and periodic transactions exist, some capability is needed to preempt an executing transaction when a higher priority transaction becomes ready to run. To maximize database system performance, however, it may not be acceptable to preempt at any time if such preemption results in having to restart or partially roll-back the preempted transaction. A more favorable approach might be to allow the preempted transaction to execute (roll-forward) if it does not prevent the higher priority transaction from meeting its deadline. This roll-forward approach could allow a transaction to execute to completion or reach a point where preemption can safely occur. Such an approach would not only be more efficient, but may also result in more transactions meeting their deadlines.

Deadlock. Another important issue in real-time transaction scheduling is transaction deadlock. In real-time databases, transactions cannot be allowed to deadlock with other transactions. If deadlocks are allowed, no reasonable guarantee can be made about meeting a transaction's deadline. Even if an adequate deadlock detection and resolution scheme is used, the time spent recovering from deadlock may degrade system performance below a tolerable level. It is important that real-time transaction scheduling mechanisms produce schedules that are deadlock free.

Priority inversion. The final issue of importance in real-time scheduling is priority inversion [Sha88b]. Priority inversion can occur in a concurrency control protocol when a higher priority transaction requires a data item held by a lower priority transaction. One obvious solution is to preempt the lower priority transaction and allow the higher priority transaction to execute. In other words, we could roll-forward or roll-back the transaction to a safe state and preempt it. In database systems where rolling back or restarting a transaction requires recovery action on the database, this simple approach is not good enough because a chain of blockage can occur.

Chained blockage is an execution state in which several lower priority transactions hold data items required by a higher priority transaction. Chained blockage [Sha88b] occurs when transactions are given unlimited access to unlocked data items by a locking protocol and preemption is used to allow higher priority transactions to run. Such a protocol can result in an arbitrarily long chain of low priority transactions blocking a higher priority transaction.

The problem of chained blockage is illustrated by Figure 3-1. Let the order of transaction entry into the system be T1, T2, T3, and T4. Also, assume the same order for transaction priorities with T1 having the lowest priority and T4 having the highest priority. The execution state depicted in Figure 3-1 is reached by the following scenario: transaction T1 enters the system and locks data items 01 and 02. Transaction T1 is then preempted by transaction T2. Transaction T2 locks data item 03 and is then preempted by transaction T3. Transaction T3 locks data item 04 and is then preempted by transaction T4. Since transaction T4 is the highest priority transaction in the system at this time, it is allowed to run to completion. Transaction T4 immediately locks data item 05 and proceeds to accomplish its job. However, when transaction T4 attempts to lock data item 04 it has to block because this item is currently held by transaction T3. This is where priority inversion begins. If roll-back or restart is not allowed, the system must allow transaction T3 to run until it can unlock data item 04. When data item 04 is released, transaction T4 then locks it and is allowed to continue. Transaction T4 then immediately attempts to lock data item 03 and blocks. Once again, priority inversion occurs. As you can see, this chain of blockage will continue until transaction T1 releases data item 01.

T1:	T2:	T3:	T4:
lock(01)			
...			
lock(02)			
(preempted)			
	lock(03)		
	...		
	(preempted)		
		lock(04)	
		...	
		(preempted)	
			lock(05)
			....
			lock(04) = BLOCK
...			
(commit)			
			...
			lock(03) = BLOCK
	...		
	(commit)		
			...
			lock(01) = BLOCK
		...	
		(commit)	
			...
			(commit)

Figure 3-1 Schedule Illustrating Chained Blockage

The execution schedule of Figure 3-1 required the highest priority transaction in the system to wait for completion of three lower priority transactions before it could execute to completion. This situation is very undesirable in real-time databases. In order to maximize performance and provide some guarantee that a transaction will meet its deadline, priority inversion must be managed so that chained blockage is avoided or reduced in severity.

An effective real-time transaction scheduling technique must have the properties of flexibility and predictability and deal effectively with the priority management issues of preemption, deadlock and priority inversion so that as many transactions as possible finish by their deadlines.

CHAPTER 4
RELATED WORK

There have been several mechanisms developed for the scheduling and concurrent execution of real-time tasks. Each of these mechanisms can be classified as either static or dynamic. This classification is based on whether schedules are dominated by a static scheduling technique that schedules tasks before they enter the system or by a dynamic scheduling technique that considers tasks when they enter the system. The objective of static approaches is to determine a feasible schedule from a set of known tasks that do not change. In the dynamic approach, the tasks entering the system are immediately included for generating the schedule.

The purpose of this chapter is to discuss the static and dynamic approaches taken by several real-time scheduling techniques available in the literature. Each approach will be evaluated based on the desirable properties of concurrency control and real-time transaction scheduling.

Static Schedulers

The distributed real-time system of MARS (MAintainable Real-time System) [Dam89] provides a static scheduling approach for periodic tasks. Each task is designated with various starting points and stopping points. Tasks are activated by the system at their starting points. However, the system relies on each task to release the processor at its stopping point. In spite of the static nature of this mechanism, task schedules can be developed to run at different phases of system operation. In order to avoid inconsistencies when scheduling switches occur, each possible switching point must be predefined at the design-stage.

The MARS concurrency approach is inadequate for effective concurrency control in a real-time database system. Even though this approach has the properties of consistency and correctness, it lacks the property of modularity. Tasks cannot be modified without affecting the correct execution of other tasks. These limitations make the MARS scheduler inappropriate for use in a real-time database system.

The MARS scheduling approach is not flexible enough to support real-time transaction scheduling. The MARS scheduler lacks the capability to handle event-driven transactions. Deadlock and priority inversion can be prevented by defining a static schedule where these do not occur. Static preemption points can also be defined so that data item conflicts are managed appropriately and potential data inconsistencies do not occur. Without the capability of handling event-driven transactions, however, the MARS system scheduling approach lacks the flexibility necessary for use in a real-time database system.

Another distributed real-time system using a static scheduling technique is ARTS (Advanced Real-Time Technology System) [Tok89]. The rate monotonic scheduling algorithm is used by ARTS to analyze the scheduling of simple periodic tasks. Scheduling of dependent tasks is resolved by an integrated time-driven scheduler and a priority inheritance protocol. Each task is classified as being hard or soft. A hard real-time task must complete by its deadline or fatal damage is presumed to occur. A soft real-time task is not as critical and is run even if its deadline cannot be met. The hard periodic tasks are scheduled first; next, available cycles are allocated to the soft task set.

As with MARS, lack of modularity makes the ARTS concurrency control approach inappropriate for real-time databases. Changes to periodic transactions scheduled under the ARTS approach invalidates the static execution schedule. In addition, no concurrency control is available between periodic and event-driven transactions. Time slots for hard periodic transactions are allocated in advance with remaining time slots used for the soft transaction set. No on-line capability exists to avoid the concurrency problems that may arise during execution of these transactions.

The ARTS scheduling approach is also inappropriate for use in real-time databases. It is not flexible enough to ensure that event-driven transactions complete by their deadlines. In addition, the static preemption points provided by this mechanism can cause concurrency control problems for event-driven transactions. If a locking protocol is used for access to shared data items, deadlocks can form. Also, since periodic transactions are scheduled in advance, priority inversion can form between periodic and event-driven transactions.

Another static scheduling technique available in the literature is Shih et al. [Shi89]. In their approach, each task is divided into a mandatory subtask and an optional subtask. The mandatory subtask is executed to accomplish a minimal computation and the optional subtask is run to decrease the error of the computation. The objective of this protocol is to ensure each mandatory subtask will meet its deadline. Once each mandatory subtask is guaranteed to meet its deadline, the available processor cycles are allocated to the optional subtasks to minimize the average error of the mandatory subtask set. A system using this protocol provides a tradeoff of computation accuracy for that of computation timing requirements. The schedules are developed by a preemptive algorithm using only the ready times, deadlines, and processing requirements of the tasks.

This algorithm is inadequate for use in real-time databases for two reasons. First of all, no concurrency control mechanism is available to control the concurrent execution of dependent transactions. Secondly, the system is not flexible enough to ensure event-driven transactions can complete by their deadlines.

Another static approach presented in the literature, though not incorporated into a real-time system, is that of Sha et al. [Sha88b]. This approach is specifically oriented to support a real-time database system. Their scheduler is based on a nonserializable concurrency control approach [Sha88a] and a priority ceiling protocol. Each task is decomposed into a partially ordered set of elementary transactions. Atomicity is accomplished at the elementary transaction level rather than the task level. The database is also decomposed into atomic data sets having the property that each elementary transaction accessing an atomic data set maintains the consistency requirements of that atomic data set.

Their use of a nonserializable concurrency control approach directly supports preemption of tasks. Preemption is also supported by minimizing the duration of transaction blocking. When blocking does occur, a priority management technique is used to ensure that lower priority tasks do not needlessly block higher priority tasks. The concurrency control protocol used by each elementary transaction is a locking protocol called the setwise two-phase lock. Elementary transactions are restricted to accessing data from one atomic data set, because holding locks across atomic data sets increases blocking duration. The setwise two-phase lock uses priority inheritance and a priority ceiling protocol to prevent deadlocks and to minimize chain blockage to at most one elementary transaction. Since allocation of locks on data objects is based on the priority of the highest priority transaction accessing that object, this approach is constrained to be a static protocol. Event-driven tasks are handled by buffering them and treating them as periodic.

The usefulness of their nonserializable real-time concurrency control approach cannot be completely evaluated until more research is accomplished in the area of nonserializable concurrency control. However, in the situation where the transactions and database cannot be decomposed, their approach reduces to that of a two-phase locking protocol.

The only drawback of this concurrency control approach is their use of the priority ceiling protocol. This protocol places too high a restriction on the concurrent execution of transactions. This approach, however, has the properties of consistency, correctness, and modularity.

The major limitation in their approach for real-time transaction scheduling is that of flexibility. If the majority of transactions in a real-time database system are periodic, then their approach has all the desirable characteristics of an effective real-time transaction scheduler. However, in a real-time database system where many transactions are event-driven and these transactions must complete by their deadline, their approach will not adequately handle priority inversion between periodic and event-driven transactions.

Dynamic Schedulers

The MARUTI real-time operating system [Lev89] has a dynamic scheduler capable of providing a guarantee of meeting deadlines of accepted tasks. Preallocation of all required services and resources is done in order to provide this guarantee. Upon verification that the new task's deadline can be met, an allocator reserves the necessary resources. Verification of scheduling feasibility is accomplished through the use of a data structure called a calendar. Jobs with non-deterministic execution time bounds and nonreal-time jobs are executed off-line. Resource allocation is independent of job scheduling with allocation being accomplished off-line due to its unbounded execution time.

The MARUTI system concurrency control approach is inadequate to support a real-time database. Concurrency is severely reduced by the MARUTI policy of preallocating resources. Preallocation of resources, however, is a consistent, correct, and modular approach to concurrency control.

The MARUTI system scheduler is also inadequate for use in real-time databases because accepted transactions cannot be preempted by transactions being scheduled. This lack of preemption allows priority inversion to occur. Deadlocks, however, are prevented by the preallocation process.

The approach taken in the Spring kernel [Sta89] is based on the notion of predictability and on-line dynamic guarantees of deadlines. Each task in a real-time system is known and can be classified as either critical, essential, or non-essential. Selective static preallocation of resources is accomplished for critical tasks providing a static guarantee of meeting their deadlines. The objective is to ensure all critical tasks complete by their deadline and as many as possible of the other tasks complete by their deadline. The scheduling algorithm is executed off-line by a system processor. Application tasks can run on both system and application processors. If a single processor cannot guarantee the timely completion of an essential task, an attempt is made to schedule the task at another processor (distributed scheduling). Code for tasks is replicated at various nodes so that only some state information need be transmitted during distributed scheduling. The algorithm employs a heuristic function to determine feasible schedules. Resource conflicts are avoided by scheduling dependent tasks at different times. This algorithm can be extended to the case where each resource may have multiple instances.

The concurrency control approach taken in the Spring kernel is inadequate for use in a real-time database system. The Spring kernel overly restricts the concurrent execution of transactions. Dependent transactions must be executed at different times to achieve consistency and correctness. The scheduling of dependent transaction at different times, however, is a consistent, correct, and modular concurrency control approach.

The scheduling approach in the Spring kernel avoids all of the issues important in developing an effective real-time transaction scheduler. When dependent transactions are scheduled at different times, the issues of deadlock, priority inversion, and preemption are avoided. Because these issues are avoided, the Spring kernel's scheduling approach lacks the transaction management tools needed for use in a real-time database. While this scheduling approach ensures all critical tasks are accomplished by their deadlines, it restricts all critical tasks to be independent. The Spring kernel's avoidance of the issues of deadlock, priority inversion, and preemption has resulted in a scheduling approach unfit for use in real-time databases.

In the Hawk real-time system [Hol89] the scheduler simply selects the highest priority task in the ready state or currently running for execution. Deadlock prevention is accomplished by imposing a strict order on the locking of resources.

The support provided for the concurrent execution of transactions in the Hawk real-time system is very limited. The policy of placing a strict order on the locking of resources to avoid deadlock places too high a restriction on the concurrent execution of transactions. However, the Hawk system's concurrency approach does have the properties of consistency, correctness, and modularity.

The Hawk scheduler is inadequate in its effort to ensure transaction deadlines are met. The Hawk system does not deal with the problem of priority inversion. Simply selecting the highest priority task to execute is inadequate since preempted tasks may hold locks on items that this new task requires.

Another dynamic approach available in the literature is Zhao et al. [Zha87]. Their approach is similar to the approach taken in the Spring kernel [Sta89]. The approach taken by Zhao et al. generates a preemptive schedule considering both time and resource constraints of tasks. When tasks enter the system they are scheduled if possible. Scheduling is done by allocating time slices to each task such that each task is executed in one or more slices and completes by its deadline. The authors indicate that the algorithm can be modified to handle multiple resource instances.

A major drawback of this concurrency control approach is the lack of consistency and correctness. Any transaction can be preempted at any time without any regard to the impact the preemption will have. As a result, transactions cannot be guaranteed to execute correctly or leave the database in a consistent state. The authors contend that nonpreemption of selective tasks can be handled with slight modifications to the algorithm. This modification, however, will place too high a restriction on the concurrent execution of dependent transactions. Modularity is missing in this approach because transactions cannot be scheduled to execute correctly without reference to the semantics of other transactions.

The scheduling approach taken by Zhao et al. is adequate for use in a real-time database system. Deadlocks and priority inversion can be avoided by the algorithm if the dynamic preemption points are established properly.

Table 4-1 provides a comparison of each algorithm's concurrency control approach based on the desirable properties of correctness, consistency, and modularity.

Table 4-2 provides a comparison of each algorithm's scheduling technique based on the issues of deadlock, priority inversion, and preemption. Also, each mechanism is evaluated on its flexibility in handling time-driven and event-driven transactions.

The lack of modularity and flexibility for the concurrent execution and scheduling of both periodic and event-driven transactions make most existing real-time concurrency control and scheduling techniques inadequate for use in real-time databases.

Table 4-1 Comparison of Each Algorithm's Concurrency Approach

Algorithm	Concurrency Approach	Correct	Consistent	Modular
MARS	Predefined Interleavings	Yes	Yes	No
ARTS	Predefined Interleavings	No	No	No
Shi89	Predefined Interleavings	Yes	Yes	No
Sha88b	Restricted Locking	Yes	Yes	Yes
MARUTI	Preallocation of Resources	Yes	Yes	Yes
Spring Kernel	No Interleavings For Dependent Tasks	Yes	Yes	Yes
Hawk	Resource Ordering	Yes	Yes	Yes
Zha87	Dynamically Defined Interleavings	No	No	No

Most of the existing techniques are biased toward meeting the timing constraints of periodic tasks. A real-time scheduler must be flexible enough to meet the deadline of any transaction when that transaction has the highest priority of all executable transactions.

Many of these techniques avoid deadlocks and resource conflicts with too much restriction on concurrency. Other alternatives for avoiding deadlocks and resource conflicts can be used so that the impact on concurrency is minimal.

Some of these schedulers do not adequately handle preemption and priority inversion for both periodic and event-driven transactions. Priority management tools must be efficiently utilized in order to ensure the highest priority transaction can complete by its deadline regardless of whether it is periodic or event-driven.

Table 4-2 Comparison of Each Algorithm's Scheduling Technique

Algorithm	Deadlock	Priority Inversion	Preemption	Flexibility
MARS	Prevention	Static Prevention	Preemptive	Strictly TD
ARTS	Inadequate	Inadequate	Preemptive	TD, Limited ED
Shi89	Prevention	Static Prevention	Preemptive	Strictly TD
Sha88b	Avoidance	Inadequate	Preemptive	TD, Limited ED
MARUTI	Prevention	Inadequate	None	Both TD,ED
Spring Kernel	Prevention	Inadequate	None	TD, Limited ED
Hawk	Prevention	Inadequate	Preemptive	Both TD,ED
Zha87	Dynamic Avoidance	Dynamic Avoidance	Preemptive	Both TD,ED

TD: Time-driven
ED: Event-driven

CHAPTER 5
THE DATAFLOW SCHEDULER

We have seen in the previous chapter how existing real time scheduling techniques are inadequate for scheduling transactions in a real-time database system. Here I present the dataflow scheduler, a real-time transaction scheduling and concurrency control technique, that provides more concurrency and flexibility than most existing real-time schedulers. A unique aspect of this scheduler is that it uses a roll-forward approach to resolving data item *conflicts*. This technique results in more transactions meeting their deadlines over that of conventional priority based conflict resolution methods of restart and roll-back.

Conceptual Framework

I will use a database system for monitoring a communications network as an example to illustrate the application of the dataflow scheduler in a real-time database system.

A communications network is composed of several sites. Each of these sites communicate with adjacent sites by using microwave, satellite, fiber optic, or cabling equipment. Each site has a maintenance control unit that keeps track of the status of local equipment. The site maintenance control unit can also perform equipment or circuit switching.

Maintenance responsibility for the network is partitioned among the maintenance locations. Each partition is assigned one primary and one secondary maintenance location. The network manager accesses information at each maintenance location to determine network maintenance status and circuit availability. Maintenance locations dispatch maintenance personnel to correct equipment failures in the network.

Each location in the communications monitoring and control system has a local database. The union of all local databases comprises the global database. Data are accessed at each location by on-site maintenance personnel during troubleshooting of site equipment. This data are also accessed by the responsible maintenance locations to determine status of communications equipment at that site.

Transactions executing in this real-time database can be modeled by tasks executing at various sites. Tasks can be initiated in serial or in parallel. This type of transaction modeling allows the separation of database operations from nondatabase operations. It also allows transactions to perform independent operations in parallel at separate sites. Atomicity is provided at the transaction level. In other words, either all or none of the transaction's tasks accessing the database are allowed to commit.

Time-driven transactions are used to report site equipment status periodically. Figure 5-1 represents a simple periodic transaction that reads a site's equipment status and updates this information at both primary and secondary maintenance locations.

```
Transaction T1:
____________________________________
begin serial
     task A:  read(site1)
     begin parallel
          task B:  update(primary)
          task C:  update(secondary)
     end parallel
     commit
end serial
```

Figure 5-1 A simple periodic transaction that reads the status at site1 and updates both primary and secondary maintenance locations

Event-driven transactions can be used to perform various functions such as troubleshooting and equipment switch-over. Figure 5-2 represents an event-driven transaction that troubleshoots communications problems between site1 and site2. Event-driven transactions can also be used by the secondary maintenance location when the primary location fails to report a site's status at the proper time.

A simple, straight-forward approach is used to model each task in a transaction task set. Each task is restricted to accessing data at one site. A nonblocking commit protocol, the three phase commit [Ber87], can be used by transactions to prevent task blocking under site failures.

```
Transaction T2:
________________________________________
begin serial
     begin parallel
          task A:  read(site1)
          task B:  read(site2)
     end parallel
     commit
     task C:  troubleshoot and report
end serial
```

Figure 5-2 A simple event-driven transaction to troubleshoot problems occurring between site1 and site2

When a task issues a request for a data item, it is given a local copy of that item if there are no conflicts with other transactions holding a copy of that data item. In this sense, data are visualized as flowing between tasks at each site. This is where the name "dataflow scheduler" comes from. Since updates are made on the local copies, there is no database recovery necessary when a transaction aborts, or is rolled back. This is similar to shadow paging [Elm89, Kor86]. In this way, priority inversion can go unchecked without affecting the scheduler's capability of meeting tasks deadlines. Priority inversion is handled with no restriction on the concurrent execution of tasks.

Each transaction task is assumed to have an execution time equal to the execution time of its parent transaction. In this way, each site can schedule tasks under the knowledge that a task's execution time is the maximum it needs to execute and perform its commit exercise with the parent transaction.

In this model, tasks are restricted to accessing data items in exclusive mode. It is unknown whether shared access will lead to better performance in a real-time database where priority is usually given to transactions performing database updates. Exclusive access is not a limitation of the dataflow scheduler.

In order to separate scheduling overhead from transaction execution, scheduling decisions can be made during task execution on a system processor.

Priority Management

The dataflow scheduler uses two levels of priority in selecting tasks for execution. First, tasks are given "hard" priorities based on how critical it is that they complete by their deadlines. Second, tasks with the same hard priority are executed with the earliest deadline first. Tasks of equal priority and deadline are executed on a first-come-first-served basis. In this manner, the highest priority task at a site in the system will be the one with the highest hard priority and the earliest deadline.

When an executing task blocks due to a resource conflict with a lower priority task, two different actions can be taken. A determination is made as to whether the lower priority task can complete without preventing completion of the higher priority task by its deadline. The lower priority task is allowed to continue execution (rolled forward) if the higher priority task can meet its deadline afterwards. Otherwise, the lower priority task is rolled-back to a "safe" point with the higher priority task continuing. This safe point occurs immediately before the point where the rolled-back task acquired the conflicting data item.

Tasks that have completed execution but have not received notification from their parent transaction to commit by the expiration of their deadline are terminated. A task that has completed execution and is uncommitted can be rolled back if it is preventing a higher priority task from meeting its deadline.

The resources required to implement the dataflow scheduler are simple. The transaction compiler must generate a task execution table for each task of a transaction. In addition, each site must maintain a data item allocation table for the data items at that site.

A sample task execution table is shown in Figure 5-3. This table lists each data item required by the task in the order it will be requested along with the location in the task where the request is made. An attribute of "time-spent" is assigned to each data item by the scheduler when the task makes and is granted a request for that data item. In addition, each task has an execution time, entry time, deadline, and hard priority included in the table.

Task1:		item	location	time-spent
execution time:	010	020	0000	---
entry:	000	060	0001	---
deadline:	025	090	00AA	---
priority:	002	010	00AB	---

Figure 5-3 Sample Task Execution Table

Each site scheduler must maintain a data item table to keep track of data items assigned to tasks. A sample data item table is shown in Figure 5-4. When a copy of a data item is given to a task, that task's identification is entered into the data item table location corresponding to that particular data item. Modifications to this table will need to be made to handle shared access modes.

Item	Task
010	Task1
030	Task4
070	Task5

Figure 5-4 Sample Site Data Item Table

The task execution tables for local tasks and the data item table are accessed and updated at each site by the local dataflow scheduler.

The task execution tables of Figure 5-5 are used to illustrate the priority management technique of the dataflow scheduler. The passage of time is represented by unit transitions in the system real-time clock. Locations in the task where resources are requested are presented in hexadecimal code.

Task1:		item	location	time-spent
execution time:	010	020	0000	---
entry:	000	060	0001	---
deadline:	025	090	00AA	---
priority:	002	010	00AB	---

Task2:		item	location	time-spent
execution time:	005	005	0000	---
entry:	005	020	0001	---
deadline:	015			
priority:	002			

Task3:		item	location	time-spent
execution time:	050	025	0000	---
entry:	010	065	0001	---
deadline:	094	060	0002	---
priority:	002	020	0130	---
		090	0C21	---
		095	0C22	---

Task4:		item	location	time-spent
execution time:	025	060	0000	---
entry:	020	020	00F3	---
deadline:	060	080	0200	---
priority:	003			

Figure 5-5 Set of Task Tables used to Illustrate the Dataflow Protocol Priority Management Technique

Task1, with a deadline of 025 and a hard priority of 002, enters the site at 000 real-time. Since it is the only task, it is immediately selected to execute. At real-time 005, task2 enters the system. Since task2 has a deadline of 015 it is higher in execution priority than the currently running task. Task2 is therefore selected to run. However, task2 immediately blocks when requesting data item 020 held by task1. A decision must now be made to either roll back task1 or allow it to complete. Figure 5-6 represents the status of task1 at the time it is preempted by task2. From Figure 5-6, we can see that task1 can complete by 010 real-time, if immediately selected to run. A check of the site data item table will indicate no conflicts exists between task1 and other tasks at the site. Based on this information, task1 is allowed to run.

Task1:		item	location	time-spent
execution time:	010	020	0000	000
entry:	000	060	0001	000
deadline:	025	090	00AA	005
priority:	002	010	01A0	---

Figure 5-6 Status of Task1 When Task2 Enters the System

When task1 completes, task2 and task3 are in the system. Since task2 has the earliest deadline, it is selected to run. Upon completion of task2, task3 is selected to run.

Task4 enters the system at 020 real-time. Since task4 has the highest hard priority with respect to task3, task3 is preempted to allow task4 to run. Figure 5-7 represents the status of task3 at the time it is preempted by task4. When task4 begins to execute it blocks because task3 holds data item 060. At this point, the scheduler must decide to either roll back task3 or allow it to complete. A check of the remaining execution time of task3 reveals that allowing it to complete would mean that task4 will not meet its deadline. As a result, task3 is then rolled back to the point where it requested data item 060.

Task4 now executes and completes at 045 real time. Task3 is then selected to run and, assuming no other task enters the system, will complete by its deadline at 091 real-time. Notice that the effect of rolling back task3 to the point it requested data item 060 resulted in having task3 complete at 091 real time. Restarting task3 would have resulted in it completing at 095 real time--past its deadline.

Task3:		item	location	time-spent
execution time:	050	025	0000	000
entry:	010	065	001A	002
deadline:	094	060	0031	004
priority:	002	020	0130	---
		090	0C21	---
		095	0C22	---

Figure 5-7 Status of Task3 When Task4 Enters the System

Algorithms for processing new task arrivals and resolving data item conflicts between tasks are shown in Figure 5-8.

```
PROCEDURE process_new_arrival

BEGIN
IF new_task_deadline < current_task_deadline AND
   new_task_priority >= current_task_priority THEN
     execute(new_task)
ELSE
     defer(new_task)
END

PROCEDURE resolve_data_item_conflict

BEGIN
IF (time_left_conflicting_task + clock +
exec_time_current_task < deadline_current_task) AND
     other_conflicts(conflicting_task) = FALSE THEN
     execute(conflicting_task)
ELSE
     roll_back(conflicting_task)
END
```

Figure 5-8 Algorithms for Processing New Task Arrivals and Resolving Data Item Conflicts

Since scheduling decisions are made off-line, the dataflow scheduler need not wait for data item conflicts to occur in order to decide how to handle a particular occurrence. By using the task execution table and data item table, the dataflow scheduler can investigate in advance any conflict where a scheduling decision must be made.

Properties

The concurrency control approach in the dataflow scheduler is consistent, correct, and modular. Database consistency is not violated since each task is executed serializably with respect to other tasks. Even though transactions can run tasks in parallel, atomicity is at the transaction level because each transaction task accessing the database commits as a whole. Since all task schedules are serializable and thus correct, the concurrency control protocol is correct. Finally, concurrency control in the dataflow scheduler is modular because tasks can be modified and new tasks can enter the system without affecting the correct execution of each task.

An important characteristic of the dataflow scheduler is that it is deadlock free. A higher priority task is only required to wait when execution of a lower priority task does not prevent it from meeting its deadline. Otherwise, conflicting tasks are rolled back to allow the priority task to continue. This approach eliminates the circular wait condition [Pet85] necessary for deadlocks to occur.

The dataflow scheduler is also flexible and predictable. This dynamic scheduling protocol gives equal consideration to both periodic and event-driven transactions based solely on the transaction's priority. A known transaction set can be simulated under the scheduler to determine its scheduling feasibility. Tables 5-1 and 5-2 provide a summary of the concurrency control and scheduling characteristics of the dataflow scheduler.

Table 5-1 Concurrency Control Characteristics of the Dataflow Scheduler

Concurrency Approach	Correct	Consistent	Modular
Two-phase locking	yes	yes	yes

Table 5-2 Scheduling Characteristics of the Dataflow Scheduler

Deadlock	Priority Inversion	Preemption	Flexibility
Avoidance	Controlled With No Performance Degradation	Preemptive	Both TD, ED

TD: Time-driven, ED: Event-driven

Failure Recovery

Under the dataflow scheduler, recovery from task failure or premature termination is easy. Since tasks have their own personal copy of each data item granted, all updates are performed on these copies. A failed task requires no subsequent database recovery. A task waiting for a data item held by a failed transaction is given a "fresh" copy from the database. Under the dataflow scheduler, the system is not burdened with the overhead of undoing work accomplished by a failed or prematurely terminated task.

Problems can occur if there is a failure at the system processor executing the dataflow scheduler. However, the system can recover from such failures by checkpointing the task execution tables and the data item table whenever a task commits. All processing accomplished by uncommitted tasks will be lost if the scheduler fails. When the scheduler is brought back up, the data item table and each task execution table are cleaned and each task restarted with fresh copies of data items from the database.

Simulation Results

A simulation program was developed to compare the performance of the dataflow scheduler's roll-forward approach to data item conflict resolution to that of rolling back or restarting a task. The simulation program, written in Pascal, is provided in Appendix A. Modifications to the program to simulate scheduling under the restart and roll-back methods are listed in Appendix B.

Three sets of 20 tasks each (15 Time-driven and 5 Event-driven with randomly generated characteristics) termed task set A, task set B, and task set C were executed by the simulation program. Tables 5-3, 5-4, and 5-5 lists the characteristics of task sets A, B, and C respectively. The individual task parameters for each task set is provided in Appendix C. Task priorities were selected randomly and vary from 0 to 9. Resource requirements were selected randomly and vary from 1 to 8. Execution times were also selected randomly and vary from 1 to 16. Deadlines are twice the execution time. The maximum resource objects in the simulation were 20. The amount of execution time surpassed at the point where all resources had been acquired is indicated by the TARA attribute.

Table 5-3 Characteristics of Task Set A

	MAX	MIN	AVG
resources	7	1	4.1
priority	9	1	5.6
execution time	15	1	8.2
period (TD)	8343	485	4025
arrivals (ED)	18086	0	10334
TARA	8	0	2.6

TARA: Time All Resources Acquired

Table 5-4 Characteristics of Task Set B

	MAX	MIN	AVG
resources	7	2	4.3
priority	9	1	4.5
execution time	15	1	9.2
period (TD)	8343	87	4246
arrivals (ED)	16143	628	8422
TARA	8	0	3.1

TARA: Time All Resources Acquired

Table 5-5 Characteristics of Task Set C

	MAX	MIN	AVG
resources	8	1	5.0
priority	9	0	4.6
execution time	15	2	8.1
period (TD)	3711	241	2073
arrivals (ED)	17221	788	10144
TARA	7	0	2.2

TARA: Time All Resources Acquired

The simulation results for each of the three task sets are shown in Tables 5-6, 5-7, and 5-8.

Figures 5-9, 5-10, and 5-11 show the relationship between the number of missed deadlines and task completions for the task sets A, B, and C respectively. Simulation run-time was selected at 8000, 10,000, 14,000, 18,000 and 22,000 seconds in order to increase the number of task completions. For each task set, the dataflow scheduler's performance exceeded that of the other two methods.

Since each scheduling technique provides a different execution sequence, a check was made to see if a particular method's sequence encountered a smaller number of preemptions or data item conflicts. Figures 5-12, 5-13, and 5-14 compares the number of preemptions and conflicts encountered by each method for task sets A, B, and C respectively. The variation of preemptions and conflicts was very slight and therefore insignificant in improving the performance of one method over that of another.

Table 5-6 Simulation Data for Task Set A

completed tasks	72	91	131	192	212	METHOD:
missed deadlines	16	20	28	37	44	RESTART
preemptions	15	19	28	36	43	
conflicts	14	18	26	35	42	
missed deadlines	16	20	27	38	42	ROLL BACK
preemptions	15	19	28	40	43	
conflicts	14	18	27	39	43	
missed deadlines	2	2	3	4	4	DATAFLOW SCHEDULER
preemptions	15	19	28	40	43	
conflicts	14	18	26	38	42	

Table 5-7 Simulation Data for Task Set B

completed tasks	155	195	275	359	437	METHOD
missed deadlines	13	17	19	24	29	RESTART
preemptions	19	22	27	32	35	
conflicts	13	16	18	23	25	
missed deadlines	13	16	18	23	28	ROLL BACK
preemptions	19	22	27	32	35	
conflicts	14	17	19	24	26	
missed deadlines	4	4	6	10	13	DATAFLOW SCHEDULER
preemptions	21	24	29	34	37	
conflicts	17	20	22	27	29	

The simulation program did not monitor the number of tasks of each priority that missed their deadlines. It is not expected that reducing the total number of missed deadlines will result in a significant increase in the number of higher priority tasks that miss their deadlines. This is an area of further research.

Table 5-8 Simulation Data for Task Set C

completed tasks	131	158	185	241	294	METHOD
missed deadlines	3	5	5	7	9	RESTART
preemptions	4	7	8	12	14	
conflicts	3	4	4	4	4	
missed deadlines	3	5	5	7	9	ROLL BACK
preemptions	4	7	8	12	14	
conflicts	3	4	4	4	4	
missed deadlines	3	4	4	6	8	DATAFLOW SCHEDULER
preemptions	4	6	7	11	13	
conflicts	3	4	4	4	4	

The dataflow scheduler seemed to be fueled by the occurrences of conflicts. In executing task set A, where the number of conflicts nearly equalled the number of preemptions, the number of missed deadlines totaled 4 from 212 completed tasks and 43 preemptions. When the number of conflicts were low, the restart and roll back algorithms performed nearly as well as the dataflow scheduler.

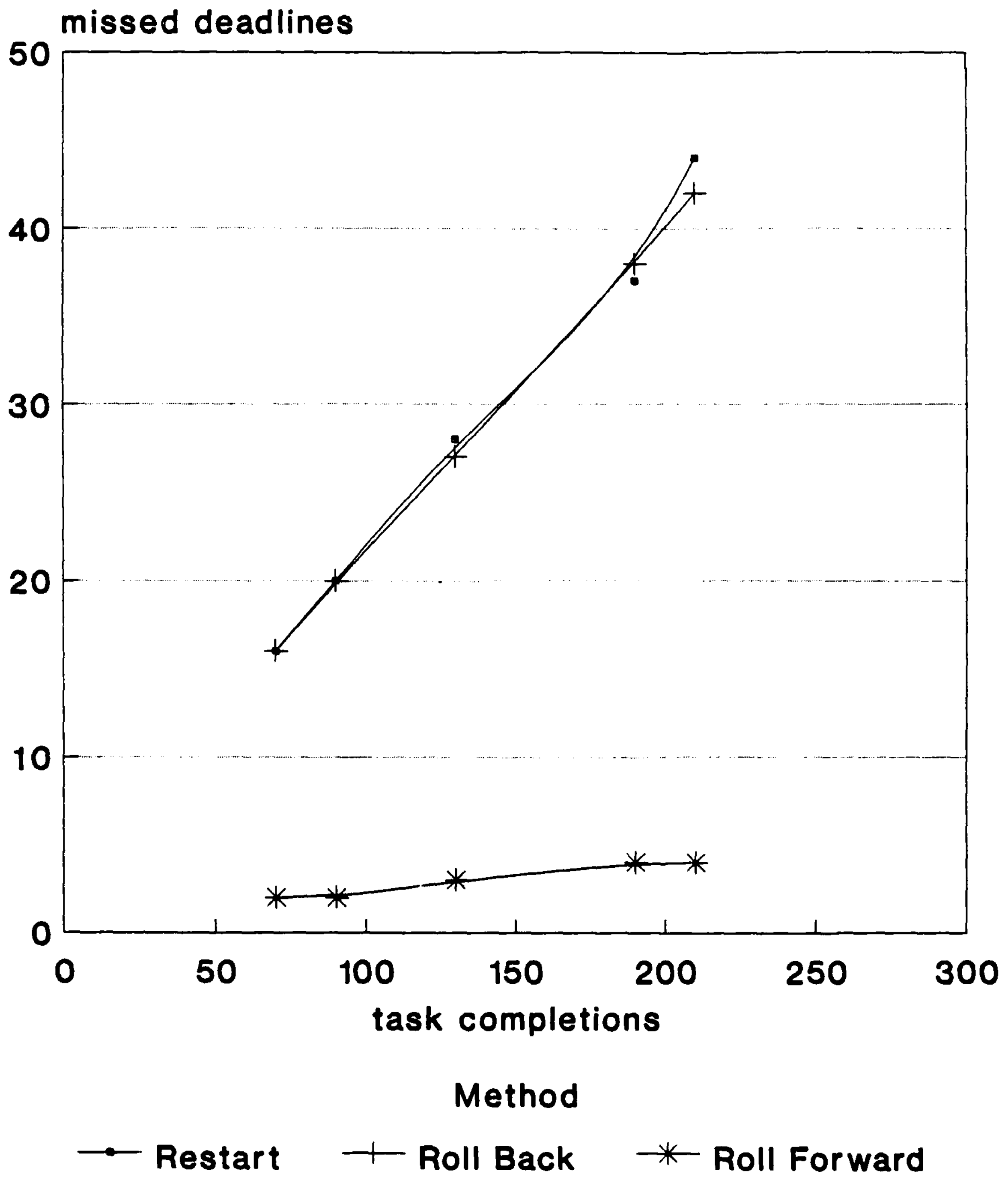

Figure 5-9 Missed Deadlines For Task Set A

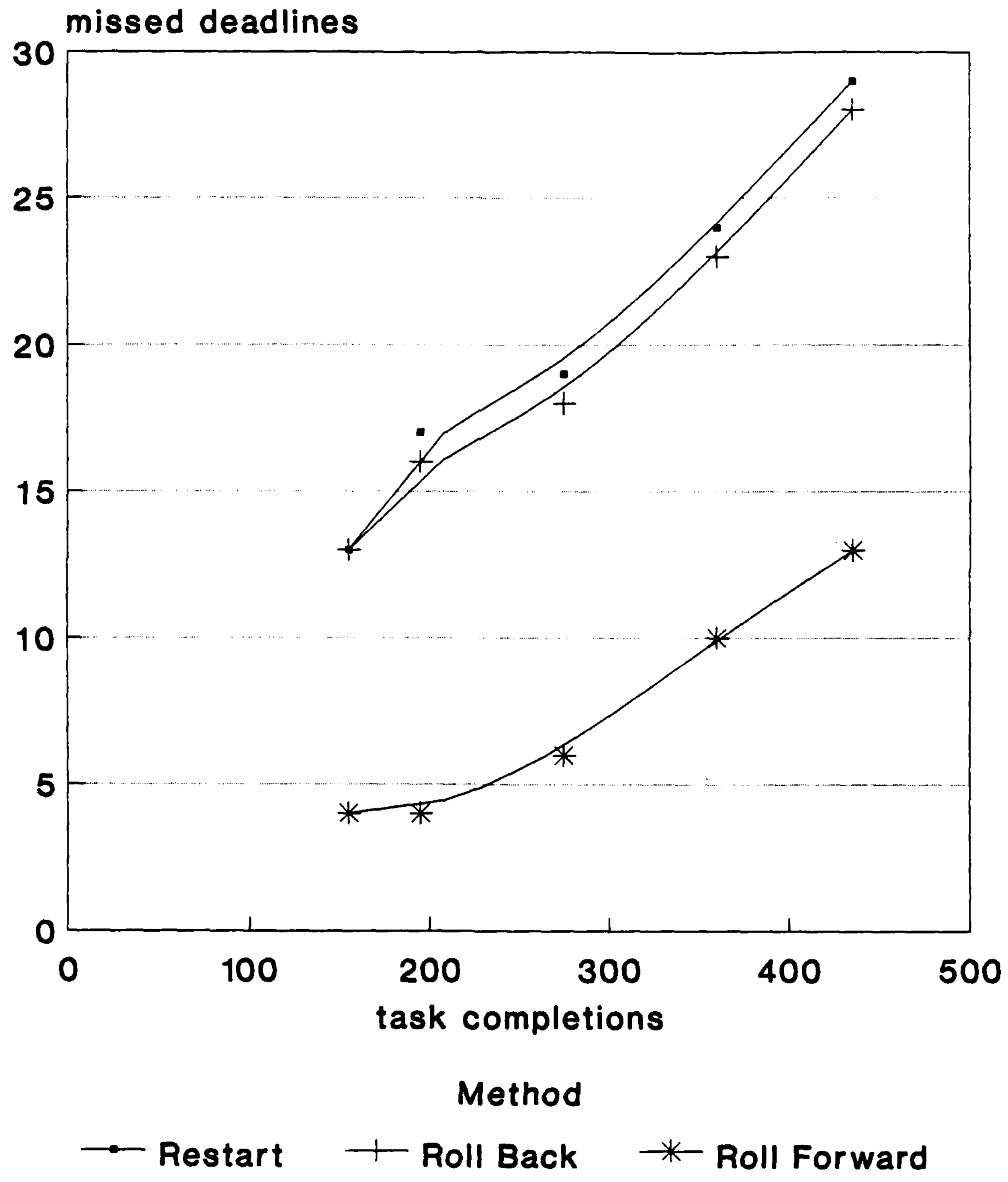

Figure 5-10 Missed Deadlines For Task Set B

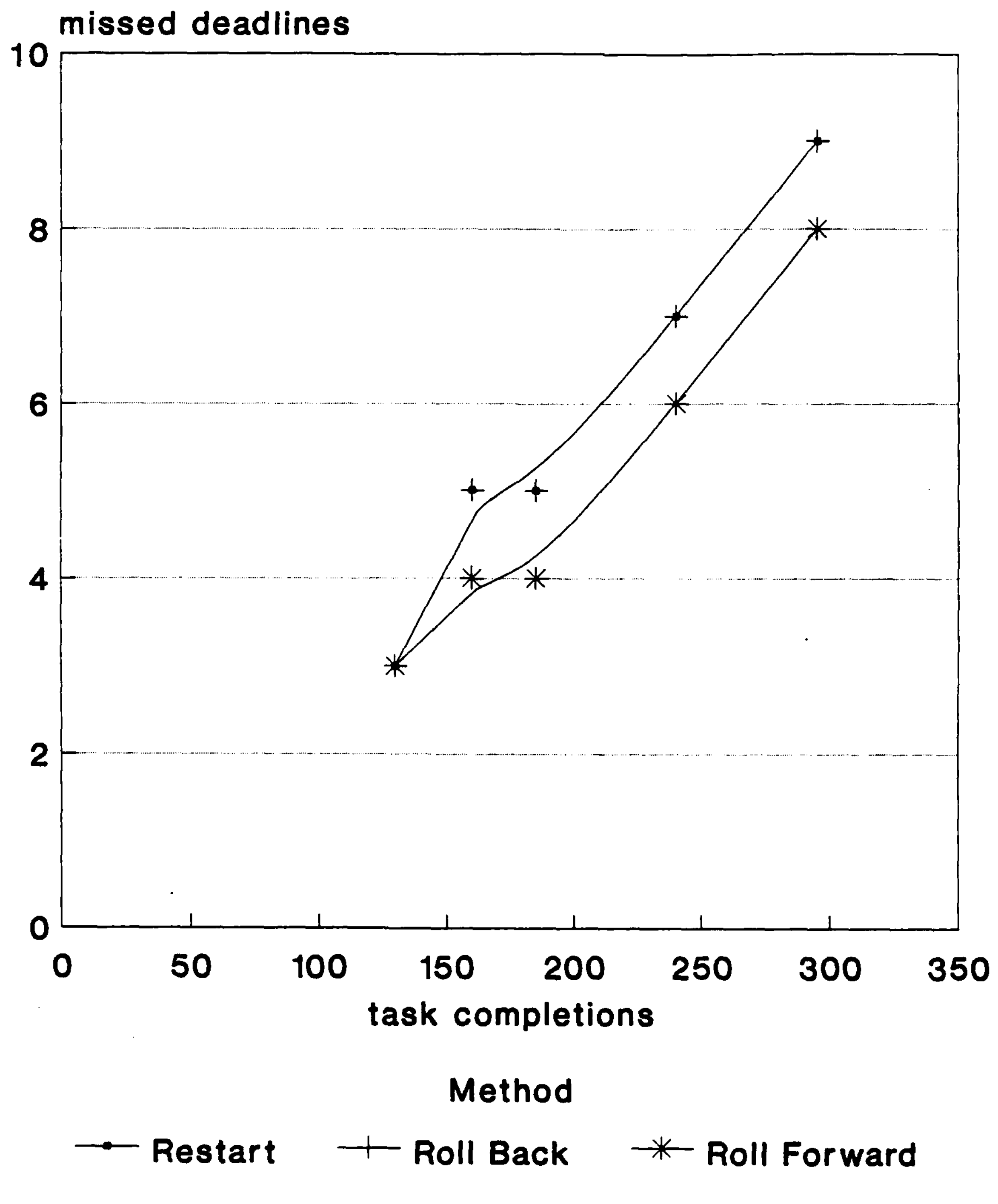

Figure 5-11 Missed Deadlines For Task Set C

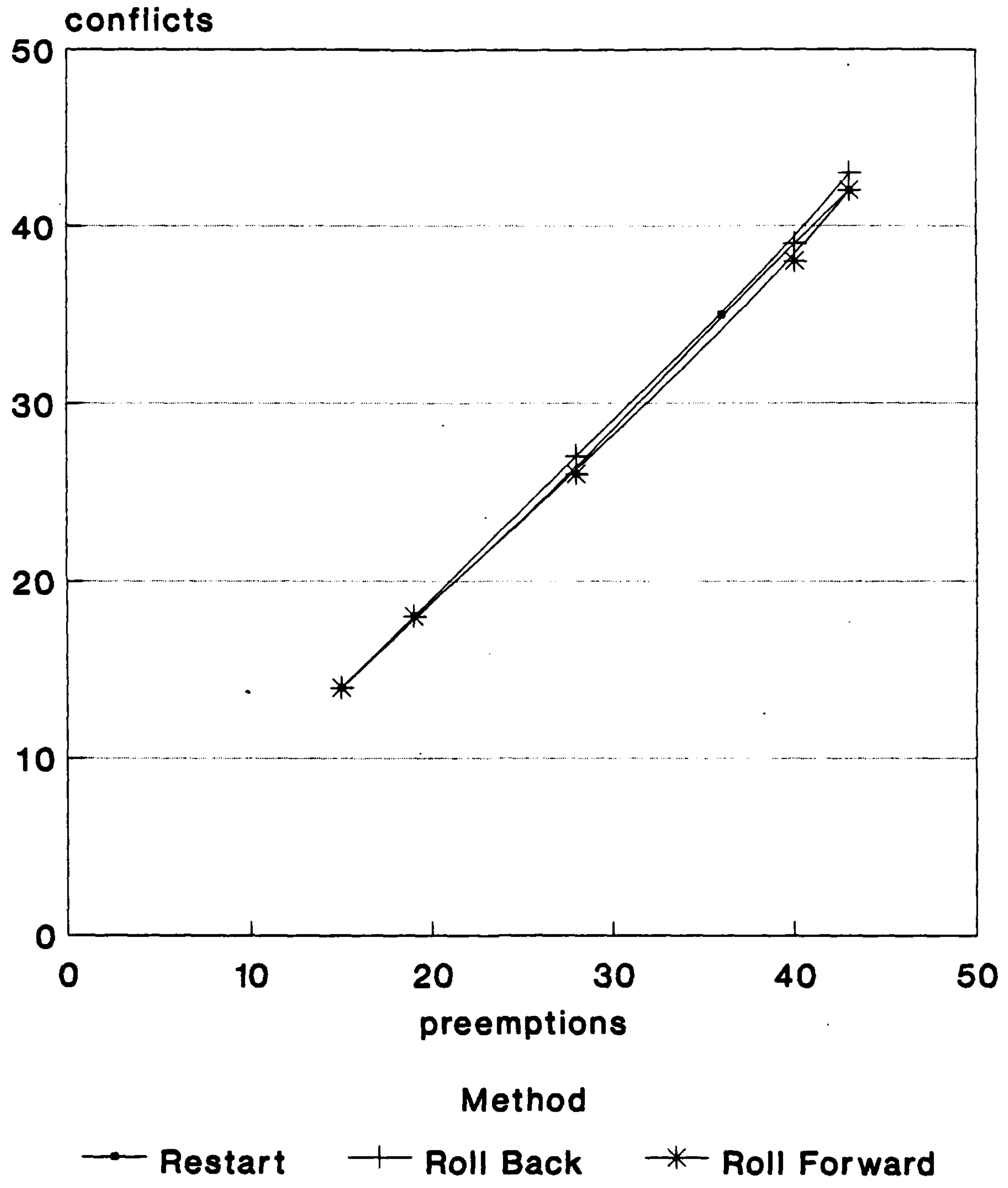

Figure 5-12 Conflicts and Preemptions For Task Set A

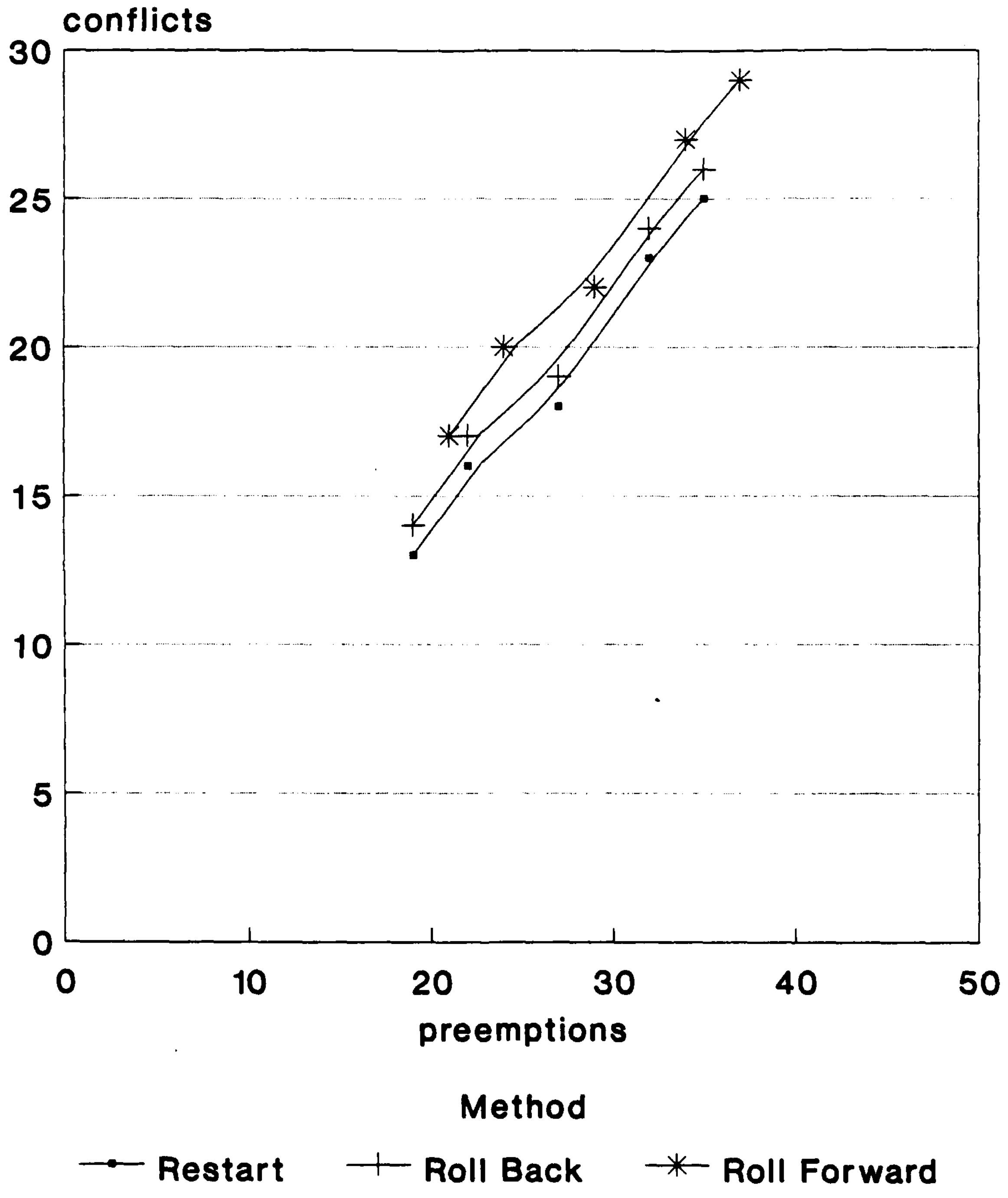

Figure 5-13 Conflicts and Preemptions For Task Set B

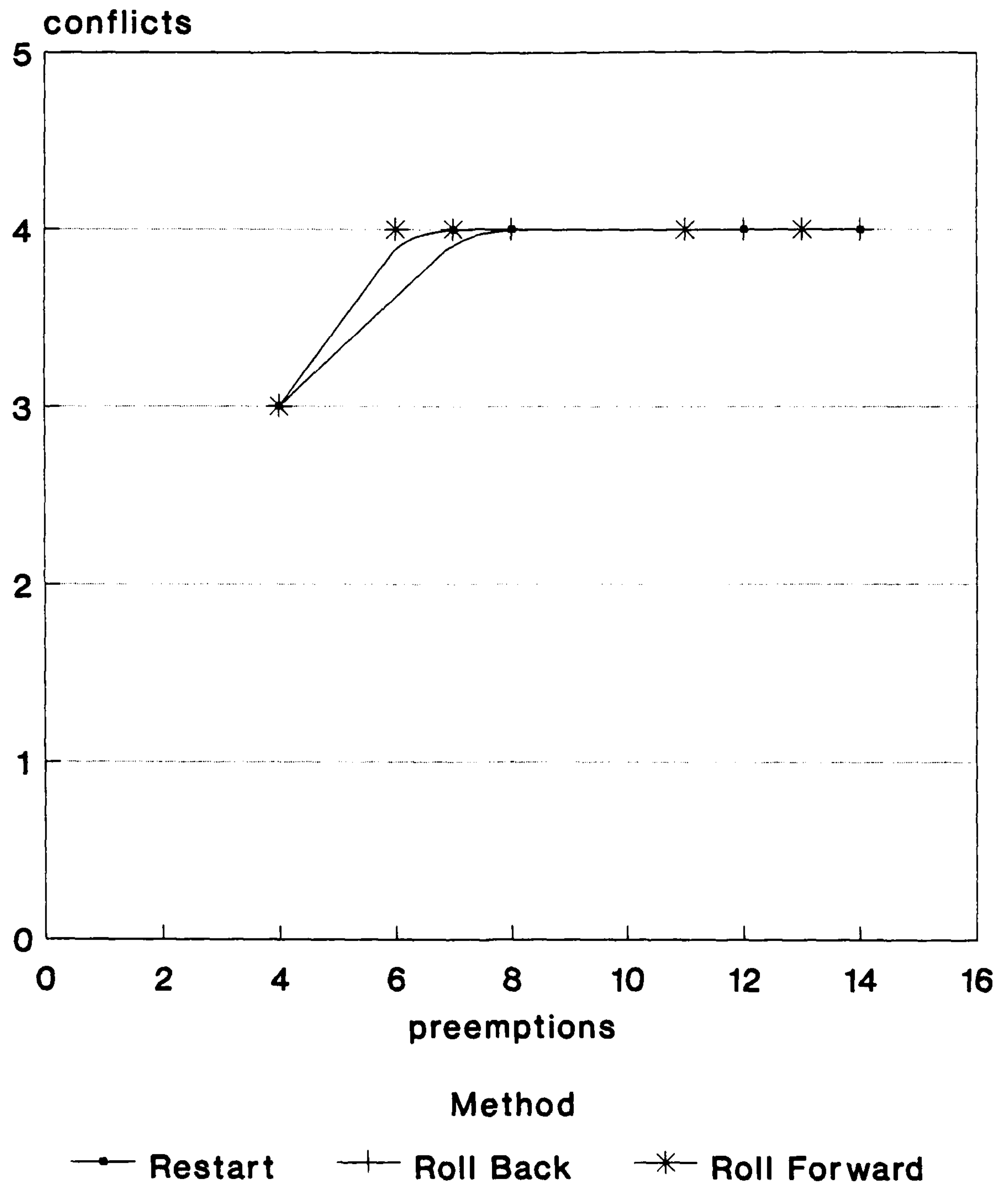

Figure 5-14 Conflicts and Preemptions for Task Set C

The roll-back method did not perform significantly better than that of restart. This can be attributed to the fact that each task accessed all of its data items by the first one-third of its execution time. In this manner, any roll-back of a task was almost equivalent to that of restarting the task. The roll-back method will probably perform much better than restart if tasks were to access data items over their entire execution time. Tasks meeting this characteristic were not simulated.

Discussion and Analysis

The dataflow scheduler has all the desirable properties of a concurrency control and a real-time transaction scheduling mechanism. The priority management technique used in the dataflow scheduler ensures freedom from deadlock. The scheduler is also flexible. Any transaction, whether periodic or event-driven can be guaranteed to complete by its deadline if its total computation time allows this. This guarantee can be made at the design stage by considering the worst case arrivals of the event-driven transactions. In this manner, any task can be classified as critical and given the highest priority. Transactions can be modified and new transactions can enter the system without impacting the dataflow scheduler. This property of modularity allows a higher degree of flexibility than existing real-time scheduling techniques. This scheduling approach is also predictable. The interaction of known transactions can be simulated early in the design stage to determine if system performance requirements will be met.

One might argue that the dataflow scheduler is too flexible for use in a "hard" real-time database system. Because the dataflow scheduler is predictable, the execution of known transactions can be simulated under this technique. Thus some degree of confidence that transactions will complete by their deadlines can be attained at the system design stage. Also, the dataflow scheduler can easily be extended to support the parallel execution of transaction tasks at each site. In this manner, sites with heavy or unpredictable loads can be provided with more processing power.

Starvation can occur when using the dataflow scheduler. As with any priority based scheduling mechanism, this can be avoided by increasing the priorities of waiting transactions. The database designer can also perform a worst-case analysis during the design stage and take actions to ensure starvation does not occur.

One drawback of the dataflow scheduler is that its use of a roll-forward approach to resolve the order of execution of conflicting transactions may not be beneficial in every case. To illustrate this point, consider transactions T3 and T4 of Figure 5-15. Tasks A and B of transaction T3 access information at the primary and secondary maintenance locations respectively, and task C updates the network manager's status of the corresponding maintenance partition in the communications network. Task D of transaction T4 accesses equipment status at site1 and tasks E and F updates the primary and secondary maintenance location respectively. Let the priority of transaction T3 be greater than that of T4.

```
Transaction T3:
begin serial
 begin parallel
  task A: read(primary)
  task B: read(secondary)
 end parallel
task C: update(network)
commit
end serial
```

```
Transaction T4:
begin serial
task D:  read(site1)
 begin parallel
  task E: update(secondary)
  task F: update(primary)
 end parallel
commit
end serial
```

Figure 5-15 Example Transactions T3 and T4

If task A begins execution at the primary maintenance location before task F and task E begins execution at the secondary maintenance location before task B, then the dataflow scheduler's roll-forward approach may result in the unnecessary execution of task A. When task F becomes ready to execute at the primary maintenance location it will preempt task A. If there is a data item conflict between tasks A and F, the system will allow task A to execute to completion if it does not prevent task F from completing by its deadline. This will result in the wasted execution of task A. Because atomicity is at the transaction level, task A cannot commit until task B does and task B will not execute because the higher priority task E began execution at the secondary maintenance location before task B became ready.

Another drawback of the dataflow scheduler is that higher priority tasks may not meet their deadlines after allowing a lower priority task to execute ahead of them. This occurs when a higher priority task allows a lower priority task to execute and is subsequently preempted by an even higher priority task. In this manner, the set of tasks that miss their deadlines may shift to the higher priority end of the spectrum of tasks.

There are two ways of handling this problem. First, the process of rolling forward a lower priority task can be restricted so that critical tasks never have to wait. Second, a worst case analysis can be accomplished at the design stage to determine those tasks that can be allowed to wait for lower priority tasks to complete.

It is unknown whether the amount of additional concurrency provided by the roll-forward approach outweighs these associated performance drawbacks. However, the dataflow scheduler is still an effective tool for scheduling tasks under resource and timing constraints.

CHAPTER 6
CONCLUSIONS AND FURTHER RESEARCH

Existing real-time concurrency control and scheduling techniques are insufficient for use in a real-time database system. A real-time database system is characterized as having both periodic and event-driven transactions. These scheduling mechanisms lack the flexibility necessary to adequately handle the timing constraints of both periodic and event-driven transactions.

A new integrated real-time concurrency control and transaction scheduling mechanism, termed the dataflow scheduler, was presented that has the flexibility needed to handle both periodic and event-driven transactions. The dataflow scheduler's technique for controlling the concurrent execution of transactions under timing constraints in a real-time database system provides more concurrency than existing real-time concurrency control protocols. Also, the dataflow scheduler's concurrency protocol has the properties of consistency, correctness, and modularity.

The dataflow scheduler is also flexible and predictable enough to support dynamic changes in a real-time database. Some existing real-time scheduling techniques are too static to support a changing environment. Other more dynamic techniques lack the necessary priority management tools to schedule both periodic and event-driven transactions to finish by their deadlines. The dataflow scheduler schedules transactions to meet their deadlines when they arrive, regardless of whether they are periodic or event-driven. Most existing mechanisms are biased toward meeting the deadlines of periodic transactions. The predictability of the dataflow scheduler allows a system manager to simulate the interaction of known transactions.

These features of flexibility and predictability make the dataflow scheduler appropriate for use in both centralized and distributed real-time database systems.

Further research on the dataflow scheduler should include the following:

(1) Performance comparisons to that of other real-time scheduling techniques in similar environments.

(2) Extending the scheduler to allow execution of independent transaction tasks on multiple processors within a single site. This capability may be beneficial at sites where execution loads are heavy or unpredictable.

(3) Extending the scheduler and corresponding data structures to allow shared access modes for data items with comparison to that of exclusive access.

(4) Implementing the scheduler in a research-oriented distributed real-time database system or other real-time system requiring transaction capabilities.

(5) Determining whether the roll-forward approach is beneficial in a distributed real-time database system where sites are unaware of what is occurring at other sites.

(6) Extending the scheduler to consider rolling forward the highest priority conflicting transaction rather than the first conflicting transaction encountered.

(7) Evaluating the restart, roll-back, and roll-forward conflict resolution methods based on the priorities of the transactions that miss their deadlines.

Other areas of research within real-time database systems include developing and implementing a flexible transaction syntax to allow a larger degree of concurrency when transactions execute independent tasks. Also, the use of nonserializable concurrency control in distributed real-time database applications requires more research.

With continued research in the areas of distributed real-time concurrency control and transaction scheduling, the knowledge necessary to implement an effective real-time database system will soon be available.

APPENDIX A
SIMULATION PROGRAM

```
PROGRAM dataflow_scheduler;

CONST
     max_time              = 20000;
     max_resource          = 20;
     max_task              = 20;
     max_priority          = 10;
     max_exec_time         = 15;
     max_num_resource      = 8;

TYPE
     task_id          = 1..max_task;
     resource_id      = 1..max_resource;
     task_class       = (time_driven, event_driven);

     event_type       = (arrival,departure,request);
     event_ptr        = ^event_node;
     event_node       = RECORD
                        event    : event_type;
                        time     : INTEGER;
                        id       : task_id;
                        next     : event_ptr;
                        END;

     resource_ptr     = ^resource_node;
     resource_node    = RECORD
                        id       : resource_id;
                        time     : INTEGER;
                        next     : resource_ptr;
                        END;

     status_ptr       = ^status_node;
     status_node      = RECORD
                        arrival       : INTEGER;
                        deadline      : INTEGER;
                        num_preempt   : INTEGER;
                        num_roll      : INTEGER;
                        location      : resource_ptr;
                        time_spent    : INTEGER;
                        time_left     : INTEGER;
                        END;
```

```
    task_ptr  = ^task_node;
    task_node = RECORD
                id                  : INTEGER;
                task_type           : task_class;
                arrival_time        : INTEGER;
                exec_time           : INTEGER;
                deadline            : INTEGER;
                priority            : INTEGER;
                resource_list       : resource_ptr;
                status              : status_ptr;
                next                : task_ptr;
                END;

VAR
    clock           : INTEGER;
    task1           : task_ptr;
    task2           : task_ptr;
    task3           : task_ptr;
    task4           : task_ptr;
    task5           : task_ptr;
    task6           : task_ptr;
    task7           : task_ptr;
    task8           : task_ptr;
    task9           : task_ptr;
    task10          : task_ptr;
    task11          : task_ptr;
    task12          : task_ptr;
    task13          : task_ptr;
    task14          : task_ptr;
    task15          : task_ptr;
    task16          : task_ptr;
    task17          : task_ptr;
    task18          : task_ptr;
    task19          : task_ptr;
    task20          : task_ptr;
    event_list      : event_ptr;
    out             : TEXT;
    item_table      : ARRAY [0..max_resource] OF INTEGER;
    queue           : task_ptr;
    first_event     : event_ptr;
    tot_preempt     : INTEGER;
    tot_roll        : INTEGER;
    tot_dead        : INTEGER;
    tot_kill        : INTEGER;
    tot_tasks       : INTEGER;
    tot_meet        : INTEGER;
    tot_rf          : INTEGER;
    df              : REAL;
    max_period      : INTEGER;
    run_time        : INTEGER;
    cc              : REAL;
```

```
PROCEDURE print_final_status;

BEGIN
WRITELN('total tasks handled is: ',tot_tasks);
WRITELN('total tasks meet deadline is: ',tot_meet);
WRITELN('total preemptions: ',tot_preempt);
WRITELN('total rollbacks: ',tot_roll);
WRITELN('total rollforward: ',tot_rf);
WRITELN('total deadline failures: ',tot_dead);
WRITELN('total killed: ',tot_kill);
WRITELN('clock is now: ',clock);
END;

PROCEDURE print_item_table;
VAR
     i    : INTEGER;

BEGIN
WRITELN(out,'item table is:');
FOR i := 0 TO max_resource DO
   WRITELN(out,'item ',i,'is ',item_table[i]);
END;

PROCEDURE print_task (task : task_ptr);

BEGIN
WRITELN(out);
IF task^.task_type = event_driven THEN
   WRITELN(out,'task type is event_driven')
ELSE WRITELN(out,'task type is time_driven');

WRITELN(out,'task id: ',task^.id);
WRITELN(out,'task arrival: ',task^.arrival_time);
WRITELN(out,'exec time: ',task^.exec_time);
WRITELN(out,'deadline: ',task^.deadline);
WRITELN(out,'priority: ',task^.priority);
END;



PROCEDURE print_task_resource (task : task_ptr);
VAR
     resource   : resource_ptr;

BEGIN
WRITELN(out);
WRITELN(out,'    id        time');
resource := task^.resource_list;
WHILE resource <> NIL DO
   BEGIN
   WRITELN(out,resource^.id:10, resource^.time:10);
   resource := resource^.next;
   END;
END;

FUNCTION resource (exec : INTEGER) : resource_ptr;

VAR
     top              : resource_ptr;
     num_resource     : INTEGER;
     temp             : resource_ptr;
     templ            : resource_ptr;
     i                : INTEGER;

BEGIN
num_resource := ROUND(random*(max_num_resource - 1)) + 1;
new(top);
top^.id := ROUND(random*max_resource/num_resource);
top^.time := 0;
temp := top;
FOR i := 2 TO num_resource DO
   BEGIN
   new(templ);
   templ^.id := ROUND(random*max_resource/num_resource) +
                temp^.id + 1;
   templ^.time := ROUND(random*exec/num_resource) +
                  temp^.time;
   temp^.next := templ;
   temp := templ;
   END;
IF num_resource = 1 THEN temp^.next := NIL
ELSE templ^.next := NIL;
resource := top;
END;


PROCEDURE put_event (task_id : INTEGER;
                     time    : INTEGER;
                     event   : event_type);
VAR
     event1    : event_ptr;
     temp      : event_ptr;

BEGIN
temp := event_list;
IF time <= temp^.time THEN
   BEGIN
   new(event1);
   event1^.event := event;
   event1^.time := time;
   event1^.id := task_id;
   event1^.next := event_list;
   event_list := event1;
   END
```

```
ELSE
   BEGIN
   WHILE (time > temp^.time) AND (time > temp^.next^.time)
      AND (temp^.next <> NIL) DO temp := temp^.next;
   new(event1);
   event1^.event := event;
   event1^.time := time;
   event1^.id := task_id;
   event1^.next := temp^.next;
   temp^.next := event1;
   END;
END;

PROCEDURE create_arrivals (task   : task_ptr);

VAR
     time : INTEGER;

BEGIN
time := task^.arrival_time;
IF (task^.task_type = event_driven) AND (time < run_time)
THEN put_event(task^.id, time, arrival);
IF task^.task_type = time_driven THEN
   BEGIN
   WHILE time < run_time DO
      BEGIN
      put_event(task^.id, time, arrival);
      time := time + task^.arrival_time;
      END;
   END;
END;

PROCEDURE initialize;

VAR
     temp      : task_ptr;
     temp1     : task_ptr;
     top       : task_ptr;
     i         : INTEGER;

BEGIN
clock := 0;
tot_preempt := 0;
tot_roll := 0;
tot_dead := 0;
tot_kill := 0;
tot_tasks := 0;
tot_meet := 0;
tot_rf := 0;
new(temp);
temp^.id := 1;
temp^.task_type := event_driven;
```

```
temp^.arrival_time := ROUND(random*max_time);
temp^.exec_time := ROUND(random*max_exec_time) + 1;
temp^.deadline := ROUND(df*temp^.exec_time);
temp^.priority := ROUND(random*max_priority);
temp^.resource_list := resource(temp^.exec_time);
top := temp;

FOR i := 2 TO max_task DO
     BEGIN
     new(temp1);
     temp1^.id := i;
     IF i <= 16 THEN temp1^.task_type := time_driven
     ELSE temp1^.task_type := event_driven;
     IF temp1^.task_type = time_driven THEN
     temp1^.arrival_time := ROUND(random*max_period)
     ELSE
     temp1^.arrival_time := ROUND(random*max_time);
     temp1^.exec_time := ROUND(random*max_exec_time) + 1;
     temp1^.deadline := ROUND(df*temp1^.exec_time);
     temp1^.priority := ROUND(random*max_priority);
     temp1^.resource_list := resource(temp1^.exec_time);
     temp^.next := temp1;
     temp := temp1;
     END;
temp^.next := NIL;
task1 := top;
task2 := task1^.next;
task3 := task2^.next;
task4 := task3^.next;
task5 := task4^.next;
task6 := task5^.next;
task7 := task6^.next;
task8 := task7^.next;
task9 := task8^.next;
task10 := task9^.next;
task11 := task10^.next;
task12 := task11^.next;
task13 := task12^.next;
task14 := task13^.next;
task15 := task14^.next;
task16 := task15^.next;
task17 := task16^.next;
task18 := task17^.next;
task19 := task18^.next;
task20 := task19^.next;

task1^.next := NIL;
task2^.next := NIL;
task3^.next := NIL;
task4^.next := NIL;
task5^.next := NIL;
task6^.next := NIL;
```

```
task7^.next := NIL;
task8^.next := NIL;
task9^.next := NIL;
task10^.next := NIL;
task11^.next := NIL;
task12^.next := NIL;
task13^.next := NIL;
task14^.next := NIL;
task15^.next := NIL;
task16^.next := NIL;
task17^.next := NIL;
task18^.next := NIL;
task19^.next := NIL;
task20^.next := NIL;
new(event_list);
event_list^.event := arrival;
event_list^.time := task1^.arrival_time;
event_list^.id := 1;
event_list^.next := NIL;

create_arrivals(task2);
create_arrivals(task3);
create_arrivals(task4);
create_arrivals(task5);
create_arrivals(task6);
create_arrivals(task7);
create_arrivals(task8);
create_arrivals(task9);
create_arrivals(task10);
create_arrivals(task11);
create_arrivals(task12);
create_arrivals(task13);
create_arrivals(task14);
create_arrivals(task15);
create_arrivals(task16);
create_arrivals(task17);
create_arrivals(task18);
create_arrivals(task19);
create_arrivals(task20);

FOR i := 0 TO max_resource DO
   item_table[i] := max_task + 1;
END;

FUNCTION pop_event : event_ptr;

BEGIN
pop_event := event_list;
event_list := event_list^.next;
END;

PROCEDURE defer (task : task_ptr);
```

```
VAR
     temp : task_ptr;
     last : task_ptr;

BEGIN
temp := queue;
last := queue;
WHILE (temp^.priority > task^.priority) AND
      (temp^.next <> NIL) DO temp := temp^.next;
IF (temp^.priority = task^.priority) AND
   (temp^.status^.deadline > task^.status^.deadline) THEN
   BEGIN
   task^.next := last^.next;
   last^.next := task;
   END
ELSE
   BEGIN
   task^.next := temp^.nex ;
   temp^.next := task;
   END;
END;

PROCEDURE gen_stat (task : task_ptr);

VAR
     temp : status_ptr;

BEGIN
new(temp);
temp^.deadline := clock + task^.deadline;
temp^.location := task^.resource_list;
temp^.num_roll := 0;
temp^.num_preempt := 0;
temp^.arrival := clock;
temp^.time_left := task^.exec_time;
temp^.time_spent := 0;
task^.status := temp;
END;

FUNCTION task_for (id : INTEGER) : task_ptr;

BEGIN
CASE id OF
   1      : task_for := task1;
   2      : task_for := task2;
   3      : task_for := task3;
   4      : task_for := task4;
   5        task_for := task5;
   6      : task_for := task6;
   7      : task_for := task7;
   8      : task_for := task8;
   9      : task_for := task9;
   10     : task_for := task10;
   11     : task_for := task11;
   12     : task_for := task12;
   13     : task_for := task13;
   14     : task_for := task14;
   15     : task_for := task15;
   16     : task_for := task16;
   17     : task_for := task17;
   18     : task_for := task18;
   19     : task_for := task19;
   20     : task_for := task20;
   END;
END;

PROCEDURE roll_back (task : task_ptr;
                     id   : INTEGER);

VAR
     stat             : status_ptr;
     temp1,temp2      : resource_ptr;

BEGIN
tot_roll := tot_roll + 1;
stat := task^.status;
stat^.num_roll := stat^.num_roll + 1;
temp1 := task^.resource_list;
WHILE temp1^.id <> id DO temp1 := temp1^.next;
temp2 := temp1;
WHILE temp2 <> NIL DO
   BEGIN
   IF item_table[temp2^.id] = task^.id THEN
      item_table[temp2^.id] := max_task + 1;
   temp2 := temp2^.next;
   END;
stat^.location := temp1;
stat^.time_left := task^.exec_time - stat^.location^.time;
stat^.time_spent := stat^.location^.time;
END;

PROCEDURE flush_item_table;

VAR
     temp : resource_ptr;

BEGIN
temp := queue^.resource_list;
WHILE temp <> NIL DO
   BEGIN
   item_table[temp^.id] := max_task + 1;
   temp := temp^.next;
   END;
```

```
END;

PROCEDURE print_status (status : status_ptr);

BEGIN
IF clock < status^.deadline THEN tot_meet := tot_meet + 1
ELSE tot_dead := tot_dead + 1;
tot_tasks := tot_tasks + 1;
END;

PROCEDURE put_next_event;

VAR
     stat : status_ptr;

BEGIN
stat := queue^.status;
IF stat^.location = NIL THEN
   BEGIN
   IF event_list = NIL THEN
      put_event(queue^.id,stat^.time_left + clock,departure)
   ELSE
      BEGIN
      IF stat^.time_left + clock <= event_list^.time THEN
      put_event(queue^.id,stat^.time_left +
clock,departure);
      END;
   END
ELSE
   BEGIN
   IF event_list^.time >= clock + stat^.location^.time THEN
      put_event(queue^.id,clock +
stat^.location^.time,request)
   END;
END;

FUNCTION task_in_queue (id : INTEGER) : BOOLEAN;

VAR
     temp : task_ptr;

BEGIN
task_in_queue := FALSE;
temp := queue;
WHILE temp <> NIL DO
   BEGIN
   IF temp^.id = id THEN task_in_queue := TRUE;
   temp := temp^.next;
   END;
END;

PROCEDURE process_arrival (id : INTEGER);
```

```
VAR
     task_stat : status_ptr;
     q_stat    : status_ptr;
     task      : task_ptr;

BEGIN
IF task_in_queue(id) THEN
   BEGIN
   tot_kill := tot_kill + 1;
   put_next_event;
   END
ELSE
   BEGIN
   task := task_for(id);
   gen_stat(task);
   IF queue = NIL THEN
      BEGIN
      put_event(id,clock,request);
      queue := task;
      queue^.next := NIL;
      END
   ELSE
      BEGIN
      IF task^.priority < queue^.priority THEN
         BEGIN
         defer(task);
         put_next_event;
         END
      ELSE
         BEGIN
         task_stat := task^.status;
         q_stat := queue^.status;
         IF (task^.priority = queue^.priority) AND
         (task_stat^.deadline >= q_stat^.deadline) THEN
            BEGIN
            defer(task);
            put_next_event;
            END
         ELSE
            BEGIN
            task^.next := queue^.next;
            queue^.next := task;
            task := queue;
            queue := queue^.next;
            tot_preempt := tot_preempt + 1;
            defer(task);
            put_event(queue^.id,clock,request);
            END;
         END;
      END;
   END;
```

```
END;

PROCEDURE process_departure;

BEGIN
print_status(queue^.status);
flush_item_table;
queue := queue^.next;
IF queue <> NIL THEN
   BEGIN
   put_next_event;
   END;
END;

PROCEDURE roll_forward (task  : task_ptr);

VAR
     templ     : task_ptr;

BEGIN
tot_rf := tot_rf + 1;
templ := queue;
WHILE templ^.next^.id <> task^.id DO templ := templ^.next;
templ^.next := templ^.next^.next;
task^.next := queue;
queue := task;
END;

FUNCTION no_conflict (task    : task_ptr) : BOOLEAN;

VAR
     temp : resource_ptr;
BEGIN
no_conflict := TRUE;
temp := task^.status^.location;
WHILE temp <> NIL DO
   BEGIN
   IF (item_table[temp^.id] <> max_task + 1) THEN
no_conflict := FALSE;
   temp := temp^.next;
   END;
END;

PROCEDURE process_request;

VAR
     q_stat    : status_ptr;
     task      : task_ptr;

BEGIN
q_stat := queue^.status;
q_stat^.time_spent := q_stat^.time_spent +
q_stat^.location^.time;
q_stat^.time_left := queue^.exec_time - q_stat^.time_spent;
IF item_table[q_stat^.location^.id] = max_task + 1 THEN
   BEGIN
   item_table[q_stat^.location^.id] := queue^.id;
   q_stat^.location := q_stat^.location^.next;
   put_next_event;
   END
ELSE
   BEGIN
   task := task_for(item_table[q_stat^.location^.id]);
   IF (no_conflict(task)) AND (task^.status^.time_left +
      clock + q_stat^.time_left < q_stat^.deadline) THEN
      BEGIN
      roll_forward(task);
      put_next_event;
      END
   ELSE
      BEGIN
      roll_back(task,q_stat^.location^.id);
      item_table[q_stat^.location^.id] := queue^.id;
      q_stat^.location := q_stat^.location^.next;
      put_next_event;
      END;
   END;
END;

PROCEDURE execute;

VAR
     event      : event_ptr;

BEGIN
WHILE event_list <> NIL DO
   BEGIN
   event := pop_event;
   clock := event^.time;
   CASE event^.event OF
      arrival      : process_arrival(event^.id);
      departure    : process_departure;
      request      : process_request;
      END;
   END;
END;

BEGIN (*---main---*)
cc := random; cc:= random;
WRITE('enter df: ');
READLN(df);
WRITE('enter max_period: ');
READLN(max_period);
WRITE('enter run_time: ');
```

```
READLN(run_time);
ASSIGN(out,'c:\tp\programs\out.dat');
REWRITE(out);
initialize;
new(queue);
first_event := pop_event;
queue := task_for(first_event^.id);
queue^.next := NIL;
clock := first_event^.time;
gen_stat(queue);
put_event(first_event^.id,clock,request);
execute;
print_final_status;
END.
```

APPENDIX B
PROGRAM MODIFICATIONS

The following procedure (PROCESS_REQUEST) should replace the corresponding procedure in the simulation program in order the simulate ROLL-BACK:

```
PROCEDURE process_request;

VAR
     q_stat     : status_ptr;
     task       : task_ptr;

BEGIN
q_stat := queue^.status;
q_stat^.time_spent := q_stat^.time_spent +
                      q_stat^.location^.time;
q_stat^.time_left := queue^.exec_time - q_stat^.time_spent;
IF item_table[q_stat^.location^.id] = max_task + 1 THEN
   BEGIN
   item_table[q_stat^.location^.id] := queue^.id;
   q_stat^.location := q_stat^.location^.next;
   put_next_event;
   END
ELSE
   BEGIN
   task := task_for(item_table[q_stat^.location^.id]);
   roll_back(task,q_stat^.location^.id);
   item_table[q_stat^.location^.id] := queue^.id;
   q_stat^.location := q_stat^.location^.next;
   put_next_event;
   END;
END;
```

The following procedures (ROLL_BACK, PROCESS_REQUEST) should replace the corresponding procedures in the simulation program ɔ simulate RESTART:

```
PROCEDURE roll_back (task : task_ptr;
                     id   : INTEGER);

VAR
```

```
     stat            : status_ptr;
     temp1,temp2     : resource_ptr;

BEGIN
tot_roll := tot_roll + 1;
stat := task^.status;
stat^.num_roll := stat^.num_roll + 1;
temp1 := task^.resource_list;
WHILE temp1 <> NIL DO
   BEGIN
   IF item_table[temp1^.id] = task^.id THEN
      item_table[temp1^.id] := max_task + 1;
   temp1 := temp1^.next;
   END;
stat^.location := task^.resource_list;
stat^.time_left := task^.exec_time - stat^.location^.time;
stat^.time_spent := stat^.location^.time;
END;

PROCEDURE process_request;

VAR
     q_stat    : status_ptr;
     task      : task_ptr;

BEGIN
q_stat := queue^.status;
q_stat^.time_spent := q_stat^.time_spent +
                      q_stat^.location^.time;
q_stat^.time_left := queue^.exec_time - q_stat^.time_spent;

IF item_table[q_stat^.location^.id] = max_task + 1 THEN
   BEGIN
   item_table[q_stat^.location^.id] := queue^.id;
   q_stat^.location := q_stat^.location^.next;
   put_next_event;
   END
ELSE
   BEGIN
   task := task_for(item_table[q_stat^.location^.id]);
   roll_back(task,q_stat^.location^.id);
   item_table[q_stat^.location^.id] := queue^.id;
   q_stat^.location := q_stat^.location^.next;
   put_next_event;
   END;
END;
```

APPENDIX C
SIMULATION TASK SETS

The following tables contain the values of the individual task parameters (period/arrival, execution time, deadline, and priority) for the simulation task sets A, B, and C.

Table C-1 contains the parameter values of the time-driven tasks for task set A; Table C-2 contains the parameter values for the Event-driven tasks of task set A. Table C-3 contains the parameter values of the time-driven tasks of task set B; Table C-4 contains the parameter values of the event-driven tasks of task set B. Table C-5 contains the parameter values of the time-driven tasks of task set C; Table C-6 contains the parameter values of the event-driven tasks of task set C.

Table C-1 Parameters for the Time-driven Tasks of Task Set A

Task ID	Period	Execution Time	Deadline	Priority
2	1456	7	2	4
3	538	5	14	9
4	2965	8	16	2
5	4507	1	2	6
6	5856	13	26	7
7	773	13	26	5
8	7058	10	20	9
9	2413	4	8	6
10	538	3	6	8
11	6057	4	8	6
12	6230	15	30	3
13	5362	14	28	7
14	485	9	18	7
15	8343	5	10	7
16	7800	9	18	7

Table C-2 Parameters for the Event-driven Tasks of Task Set A

Task ID	Arrival	Execution Time	Deadline	Priority
1	0	1	2	9
17	18086	9	18	3
18	3608	15	30	5
19	16143	10	20	1
20	13833	9	18	1

Table C-3 Parameters for the Time-driven Tasks of Task Set B

Task ID	Period	Execution Time	Deadline	Priority
2	738	8	16	1
3	2220	13	26	3
4	7445	1	2	1
5	87	13	26	7
6	773	13	26	5
7	7058	10	20	9
8	2413	4	8	6
9	538	3	6	8
10	6057	4	8	6
11	6230	15	30	3
12	5362	14	28	7
13	485	9	18	7
14	8343	5	10	7
15	7800	9	18	7
16	8139	9	18	3

Table C-4 Parameters for the Event-driven Tasks of Task Set B

Task ID	Arrival	Execution Time	Deadline	Priority
1	628	14	28	2
17	3608	15	30	5
18	16143	10	20	1
19	13833	9	18	1
20	7899	6	12	1

Table C-5 Parameters for the Time-driven Tasks of Task Set C

Task ID	Period	Execution Time	Deadline	Priority
2	1472	13	26	3
3	3497	5	10	8
4	824	11	22	6
5	2821	16	26	8
6	3509	3	6	3
7	2439	12	24	7
8	3182	13	26	4
9	241	4	8	7
10	1260	11	22	7
11	1887	7	14	1
12	2694	2	4	9
13	1225	2	4	1
14	1951	2	4	9
15	3711	9	18	3
16	389	12	24	1

Table C-6 Parameters for the Event-driven Tasks of Task Set C

Task ID	Arrival	Execution Time	Deadline	Priority
1	17221	4	8	3
17	788	4	8	0
18	16352	10	20	3
19	7218	6	12	6
20	9139	15	30	2

REFERENCES

[Ber81] Bernstein, P. A., and Goodman, N., "Concurrency Control in Distributed Systems," ACM Computing Surveys 13 2(June 1981) 185-222.

[Ber87] Bernstein, P. A., Hadzilacos, V., and Goodman, N., Concurrency Control and Recovery in Database Systems, Addison-Wesley, Reading, Mass., 1987.

[Cer84] Ceri, S., and Giuseppe, P., Distributed Databases Principles and Systems, McGraw-Hill, New York, 1984.

[Cou88] Coulouris, G. F., and Dollimore, J., Distributed Systems Concepts and Design, Addison-Wesley, Wokingham England; Reading, Mass., 1988.

[Dam89] Damm, A., Reisinger, J., Schwabl, W., and Kopetz, H., "The Real-Time Operating System of MARS," Operating Systems Review 23 3(July 1989) 141-157.

[Elm89] Elmasri, R., and Navathe, S. B., Fundamentals of Database Systems, Benjamin/Cummings, Redwood City, Calif., 1989.

[Esw76] Eswaren, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L., "The Notions of Consistency and Predicate Locks in a Database System," Communications of the ACM 19 11(November 1976) 624-633.

[Gar83] Hector, Garcia-Molina, "Using Semantic Knowledge for Transaction Processing in a Distributed Database," ACM Transactions on Database Systems, 8 2(June 1983) 186-213.

[Hol89] Holmes, V. P., and Harris, D. L., "A Designer's Perspective of the Hawk Multiprocessor Operating System Kernel," Operating Systems Review 23 3(July 1989) 158-172.

[Kor86] Korth, H. F., and Silberschatz, A., Database System Concepts, McGraw-Hill, New York, 1986.

[Lev89] Levi, S-T., Tripathi, S. K., Carson, S. D., and Agrawala, A. K., "The MARUTI Hard Real-Time Operating System," Operating Systems Review 23 3(July 1989) 90-105.

[Lyn83] Lynch, N. A., "Multilevel Atomicity--A New Correctness Criterion for Database Concurrency Control," ACM Transactions on Database Systems, 8 4(December 1983) 484-502.

[Pet86] Peterson, J. L., and Silberschatz, A., Operating System Concepts, 2nd ed, Addison-Wesley, Reading, Mass., 1986.

[Sha88a] Sha, L., Lehoczky, J. P., and Jensen, E. D., "Modular Concurrency Control and Failure Recovery," IEEE Transactions on Computers 37 2(February 1988) 146-159.

[Sha88b] Sha, L., Rajkumar, R., and Lehoczky, J. P., "Concurrency Control for Distributed Real-Time Databases," SIGMOD RECORD 17 1(March 1988) 82-98.

[Shi89] Shih, W-K., Liu, J. W. S., Chung, J-Y., and Gillies, D. W., "Scheduling Tasks with Ready Times and Deadlines to Minimize Average Error," Operating Systems Review 23 3(July 1989) 14-28.

[Sta89] Stankovic, J. A., and Ramamritham, K., "The Spring Kernel: A New Paradigm for Real-Time Operating Systems," Operating Systems Review 23 3(July 1989) 54-71.

[Tok89] Tokuda, H., and Mercer, C. W., "ARTS: A Distributed Real-Time Kernel," Operating Systems Review 23 3(July 1989) 29-53.

[Zha87] Zhao, W., Ramamritham, K., and Stankovic, J. A., "Preemptive Scheduling Under Time and Resource Constraints," IEEE Transactions on Computers C-36 8(August 1987) 949-960.

BIOGRAPHICAL SKETCH

Ronnie E. Edge received his Bachelor of Science degree in electrical engineering from the University of South Carolina in 1984. Upon graduating, he attended the United States Air Force Officer Training School and received his commission on October 16, 1984.

Ronnie Edge served as a Microwave Systems Engineer at the 1842nd Electronics Engineering Group at Scott Air Force Base, Illinois, from November 1984 to February 1987. There He worked on several defense communications projects to include the Digital European Backbone (DEB) and the Transmission Monitoring and Control system (TRAMCON).

In March 1987 he was assigned to Air Force Communications Command Headquarters located at Scott Air Force Base, Illinois. There he served as Radio Systems Engineer on several strategic communications programs to include the Ground Wave Emergency Network (GWEN) and the Aircraft Alerting Communications Electromagnetic pulse upgrade (AACE). He is currently a captain in the United States Air Force and a graduate student at the University of Florida. He is pursuing a Master of Science degree in computer and information sciences under sponsorship from the Air Force Institute of Technology.

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

[illegible]

Yuan-Chieh Chow, Chairman
Professor of Computer and Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

[illegible]

Richard Newman-Wolfe
Assistant Professor of Computer and Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

[illegible]

Sharma Chakravarthy
Associate Professor of Computer and Information Sciences

This thesis was submitted to the Graduate Faculty of the Department of Computer and Information Sciences in the College of Business Administration and to the Graduate School and was accepted as partial fulfillment of the requirements for the degree of Master of Science.

December 1989

Dean, Graduate School